AMIGA

AMIGA

ISBN 3-89011-076-2

Copyright © 1987 DATA BECKER GmbH
Merowingerstraße 30
4000 Düsseldorf

Alle Rechte vorbehalten. Kein Teil dieses Buches darf in irgendeiner Form (Druck, Fotokopie oder einem anderen Verfahren) ohne schriftliche Genehmigung der DATA BECKER GmbH reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Wichtiger Hinweis:
Die in diesem Buch wiedergegebenen Schaltungen, Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle Schaltungen, technischen Angaben und Programme in diesem Buch wurden von dem Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler ist der Autor jederzeit dankbar.

Vorwort

Der Commodore Amiga wird zu Recht als eine sehr vielseitige Maschine beschrieben, deren Fähigkeiten reizen, sie in eigenen Programmen zu verwenden.

Es gibt auch schon eine Reihe Programmiersprachen, in denen sich der Amiga programmieren läßt. Eine davon ist diejenige, mit der eigentlich alles anfängt: die Maschinensprache. In ihr sind ja sicherlich der erste Compiler für diesen Rechner und nicht zuletzt einige Teile des Betriebssystems geschrieben worden.

Um den Rechner in dieser Sprache zu programmieren, benötigt man nicht nur Kenntnisse des Prozessors MC68000 und dessen Befehlsvorrat, sondern auch einige interne Dinge über den Amiga selbst. Die vielen Funktionen, die das Betriebssystem dieses Rechners bietet, sollten nämlich auch von einem Maschinenprogramm aufgerufen und bedient werden können.

Dies stellte bisher ein gewisses Problem dar. Die Dokumentationen über den Amiga sind fast ausschließlich für C-Programmierer gedacht und somit für die Maschinensprache keine große Hilfe. Da ich in der Fachhochschule Köln einen Emulator für den 6502-Prozessor auf dem Amiga entwicken sollte, mußte ich mich mit diesem Problem herumschlagen. Nach langen Tagen im Labor und Nächten zuhause an dem Rechner lege ich Ihnen hiermit eine Art neue Dokumentation vor, welche den Prozessor im Amiga und dessen Zugriff auf das Betriebssystem beschreibt.

Zuerst werden einige grundlegende Dinge wie Speicherorganisation und das 1x1 des Computers betrachtet, worauf nach einem Blick in das Innere des Amiga die Beschreibung des Prozessors folgt. An Hand einiger Beispielprogramme können Sie leicht die Programmierung dieses Prozessors nachvollziehen und dabei erlernen.

Danach werden Sie kennenlernen, wie einfach man mit solchen Programmen dem Amiga solch schöne Dinge wie Fenster und Menüs entlockt sowie Daten oder Texte in den Computer eingibt oder aus ihm herausschickt. All diese Funktionen werden durch Beispielprogramme unterlegt, welche man sammeln und so eine Bibliothek von brauchbaren Teilprogrammen erhalten kann. Mit dieser Bibliothek können Sie dann auf einfache Art und Weise schnelle, benutzerfreundliche und professionell aussehende Programme erstellen.

Ich wünsche Ihnen mit diesem Buch und dem Amiga viel Vergnügen beim Ausprobieren und Anwenden der Kenntnisse, die Sie bald erworben haben werden!

Stefan Dittrich *Gummersbach, im April 1987*

Inhaltsverzeichnis

1.	**Einführung**	**13**
1.1	Warum Maschinensprache?	13
1.2	Ein Blick in den Speicher	14
1.2.1	RAM, ROM, Hardware-Register	14
1.2.2	Bits, Bytes und Worte	16
1.2.3	Zahlensysteme	17
1.3	Aufbau des Amiga	20
1.3.1	Struktur	20
1.3.2	Speicher-Aufteilung	21
1.3.3	Alles auf einmal: Multitasking	23
2.	**Der Prozessor MC68000**	**27**
2.1	Register	27
2.2	Speicher-Adressierung	30
2.3	Betriebszustände	39
2.3.1	User- und Supervisor-Modus	40
2.3.2	Exceptions	41
2.3.3	Interrupts	45
2.3.4	Bedingungscodes	46
2.4	Der Befehlssatz	51
3.	**Die Arbeit mit Assemblern**	**67**
3.1	Der Assembler des Entwicklungs-Paketes	68
3.2	Der Profimat AMIGA	70
3.3	Der K-SEKA-Assembler	71

4. Programmierungs-Praxis: die ersten Programme ... 85

4.1 Addieren von Zahlen. ... 85
4.2 Sortieren einer Tabelle ... 89
4.3 Zahlensysteme umwandeln ... 93
4.3.1 Binär- in Sedezimalzahlen ... 93
4.3.2 Binär- in Dezimalzahlen ... 97
4.3.3 Sedezimal- in Binärzahlen ... 99
4.3.4 Dezimal- in Binärzahlen ... 102

5. Hardware-Register-Anwendung ... 105

5.1 Sondertasten abfragen ... 105
5.2 Laufende Zeit abfragen ... 107
5.3 Maus und Joystick auslesen ... 108
5.4 Bedienung der Parallelschnittstelle ... 111
5.5 Tonerzeugung ... 115
5.6 Übersicht der Hardware-Register ... 123

6. Arbeiten mit dem Amiga-Betriebssystem ... 127

6.1 Bibliotheken laden ... 127
6.2 Funktionen aufrufen ... 130
6.3 Programmvorbereitung ... 132
6.3.1 Speicher reservieren ... 132
6.3.2 Öffnen eines Fensters, einfache Art ... 134
6.4 Ein- und Ausgabe ... 139
6.4.1 Bildschirmausgaben ... 140
6.4.2 Tastatureingaben ... 146
6.4.3 Drucker-Ansteuerung ... 150
6.4.4 Serielle Ein-/Ausgabe ... 151
6.4.5 Sprachausgabe ... 151

6.5 Diskettenoperationen 157
6.5.1 Dateien öffnen 158
6.5.2 Daten lesen oder schreiben 159
6.5.3 Dateien löschen 162
6.5.4 Dateien umbenennen 163
6.5.5 Inhaltsverzeichnis auslesen 164
6.5.6 Direkter Zugriff auf Diskette 170

7. Die Arbeit mit Intuition 177

7.1 Bildschirm öffnen 178
7.2 Fenster öffnen 191
7.3 Meldungen und Abfragen: Requester 196
7.4 Ereignis-Behandlung 200
7.5 Menü-Programmierung 203
7.6 Textausgabe 221
7.7 Images 223
7.8 Borders 227
7.9 Gadgets 231
7.9.1 Boolean-Gadgets 232
7.9.2 String-Gadgets 240
7.9.3 Proportional-Gadgets 247

8. Beispielprogramme 253

8.1 Umschalten in den Supervisor-Modus 253
8.2 Exception-Programmierung 256

Anhang 261

Stichwortverzeichnis 277

1. Einführung

Bevor wir uns in die Arbeit mit der Maschinensprache stürzen, sollten wir noch einige Dinge betrachten, die zum Verständnis der Maschinenprogrammierung wichtig sind. Auch sollte mal erwähnt werden, was an ausgerechnet dieser Sprache interessant ist.

1.1 Warum Maschinensprache?

Die Maschinensprache ist eigentlich die einzige Sprache, die ein Prozessor wie der MC68000 versteht. Alle anderen Sprachen wie das beliebte BASIC oder C müssen daher in diese Sprache übersetzt werden, sei es erst bei der Ausführung (wie bei einem Interpreter, z.B. BASIC) oder vorher (mit einem Compiler, z.B. C).

Dies deutet schon auf einen Vorteil der Maschinensprache gegenüber den erwähnten Hochsprachen an: sie ist schneller. Bei Verwendung eines Interpreters muß jede Zeile immer wieder übersetzt werden, was Zeit kostet. Und die Übersetzung, die ein Compiler liefert, ist schließlich ein Maschinenprogramm, allerdings wegen der allgemeinen Art der Übersetzung nie so optimiert wie es ein reines Maschinenprogramm sein kann.

Dazu kommt, daß man für die Ausführung eines fertigen Maschinenprogramms keinen Interpreter braucht, was einen weiteren Vorteil gegenüber BASIC bedeutet.

Des weiteren ist man mit Hilfe der Maschinensprache in der Lage, wirklich alle Möglichkeiten eines Rechners auszuschöpfen, was sich gerade bei einem so vielseitigen Computer wie dem Amiga wirklich lohnt. Somit kann es durchaus notwendig sein, zu einem Programm in einer höheren Programmiersprache ein Maschinenprogramm hinzuzufügen, wenn man an irgendeine Funktion des Rechners in jener Sprache nicht herankommt.

Und schließlich soll ein Argument nicht unerwähnt bleiben: Für richtige Assembler-Enthusiasten (so wie ich), die z.B. vom 6502

oder vom 68000 eines anderen Rechners in der Assembler-Programmierung geübt sind, ist es keine Frage, welche Programmiersprache verwendet werden soll.

1.2 Ein Blick in den Speicher

Bevor man ein Programm in Maschinensprache schreiben will, muß man sich erst einmal klar sein, was dieses Programm tun soll und welche Hilfsmittel es dazu braucht. Wohl das wichtigste Hilfsmittel ist der Speicher, mit dem gearbeitet wird. Schließlich ist es unumgänglich, Daten und auch Programme zu speichern.

Ein solcher Speicher ist eine recht vielseitige Sache. Die Möglichkeiten seiner Anwendung übersteigen deutlich die für BASIC-Programmierer übliche Ansicht 'entweder reicht er oder nicht', da es einige Variationen gibt.

1.2.1 RAM, ROM, Hardware-Register

Das erste, was einem zum Begriff 'Speicher' einfällt, ist, daß man Daten reinschreiben und wieder auslesen kann. Diese Speicherart ist das RAM, das Random Access Memory, was frei übersetzt 'Speicher mit wahlfreiem Zugriff' bedeutet. Dieser Speicher besteht aus elektronischen Bauteilen, die hineingeschriebene Daten behalten können. Der Nachteil ist allerdings, daß all diese Daten weg sind, wenn der Rechner ausgeschaltet wird oder der Strom ausfällt.

Damit aber beim Einschalten des Rechners irgendetwas geschehen kann, wie z.B. die Aufforderung zum Einlegen der Kickstart-Diskette, muß ein Programm dafür im Speicher sein. Im RAM kann dies nicht liegen, also brauchen wir eine weitere, nichtflüchtige Speicherart: ein ROM.

ROM steht für 'Read Only Memory' und deutet dadurch schon an, daß es nur gelesen, nicht jedoch beschrieben werden kann. Ein solches ROM ist auch im Amiga eingebaut und sorgt für das Einladen der Kickstart-Diskette ins RAM, auf der nun die Pro-

gramme für den weiteren Ablauf stehen. In späteren Versionen des Amiga wird auch das Kickstart im ROM sein, so daß der Rechner sofort weiß, was er mit einer Workbench-Diskette anfangen kann.

Eine Variation von ROMs sind die sogenannten PROMs, die 'Programmable Read Only Memory'-Bausteine. Dies sind ROMs, welche durch spezielle Verfahren selbst programmiert bzw. beschrieben werden können. Sie sind allerdings recht selten, da sie einmal programmiert nicht mehr gelöscht werden können. Häufiger als PROMs findet man EPROMs, Erasable ROMs, welche durch Bestrahlung mit ultraviolettem Licht gelöscht werden können. Man erkennt sie an dem kleinen Fensterchen auf dem IC.

Es gibt auch schon, leider nicht im normalen Handel und schon gar nicht so billig wie RAM-Bausteine, sogenannte EEROMs, Electric Erasable ROMs. Diese Bausteine können elektrisch gelöscht werden und sind somit ähnlich wie RAM-Bausteine, verlieren jedoch beim Abschalten nicht ihren Inhalt.

Seit es den Amiga 500 gibt, gibt es auch eine weitere Speicherart, das WOM. Hierbei handelt es sich nicht etwa um das in der April-Ausgabe einer Elektronik-Zeitschrift beschriebene Write Only Memory, was nun wirklich keinen Sinn hätte, sondern um ein Write Once Memory. In einen Speicher dieser Art wird die Kickstart-Diskette eingelesen und ist dann nicht mehr beschreibbar. Allerdings ist dies nicht eine neue Art von Bausteinen, sondern ein RAM-Bereich. Die zum Beschreiben dieses RAMs notwendige Signalleitung wird lediglich nach dem Booten der Kickstart-Diskette gesperrt, so daß dann nur noch aus diesem Speicher gelesen werden kann.

Außer den RAMs und den diversen Arten von ROMs gibt es noch eine Speicherart, welche irgendwo dazwischen angesiedelt ist. Es handelt sich dabei um Speicherzellen, welche direkt mit Peripheriebausteinen rund um den Prozessor in Verbindung stehen. Diese Speicherzellen nennt man Hardware-Register, da sie die Hardware steuern. Wie man diese Register anwenden kann, finden Sie im Kapitel 'Hardware-Register-Anwendung'.

Doch nun zurück zum 'normalen' Speicher, dem RAM. Betrachten wir nun einmal genau dessen Aufbau und die Anwendung.

1.2.2 Bits, Bytes und Worte

Die allgemeine Größe, in der man den verfügbaren Speicher mißt bzw. angibt, ist Kilobyte (KByte). Ein Kilobyte sind 1024 Byte, nicht etwa 1000 Byte, wie man eigentlich annehmen kann. Diese merkwürdige Zählweise liegt an dem binär orientierten Aufbau jedes Rechners, wobei die meisten Zahlen in 2er-Potenzen angegeben werden. So auch die Kilobytes.

Um einen Speicher von 1 KByte anzusprechen, benötigt der Prozessor 10 Adreßleitungen, auf denen entweder 5 Volt oder 0 Volt liegen. Dadurch erreicht man 2^10=1024 verschiedene Kombinationen und dadurch Speicherbytes.

Ein solches Byte besteht wiederum aus 8 einzelnen Ja/Nein-Informationen, so daß sich in einem Byte 2^8=256 verschiedene Zahlen zwischen 0 und 255 speichern läßt. Diese einzelnen Informationseinheiten, welche somit auch die kleinsten Einheiten in der Informatik darstellen, nennt man Bits. Dies ist eine Kurzform von 'Binary coded Digit'.

In einem 512 KByte großen Speicher wie im Amiga sind also 2^19=524288 Bytes, das sind 4194304 Bits enthalten. Man stelle sich vor, daß man in diesem Speicher 2^4194304 verschiedene Kombinationsmöglichkeiten hat, womit klar ist, daß es immer neue Programme geben wird...

Zurück zu den Einheiten. Bits und Bytes reichen völlig aus, um einen 8-Bit-Prozessor wie den 6502 zu programmieren, da dieser nur mit Bytes arbeiten kann. Bei dem 16/32-Bit-Prozessor MC68000, der im Amiga eingebaut ist, muß man zwei weitere Datenarten kennen. Gemeint sind dabei die Worte, die aus 16 Bit bzw. 2 Byte bestehen, und die Langworte, 32 Bit breit und somit 4 Byte oder 2 Worte.

In einem Wort kann eine Zahl zwischen 0 und 65535 gespeichert werden, in einem Langwort gar 0 bis 4294967295. Mit diesen riesigen Zahlen kann der MC68000 nämlich durch einen einzigen Befehl arbeiten, also verschieben oder rechnen.

Ab und zu braucht man aber auch bei der Darstellung von Zahlen negative Werte. Da ein Bit nur 0 oder 1 sein kann und nicht -1, hat man eine Abmachung getroffen. Ein Wort, welches vorzeichenrichtig behandelt werden soll, trägt in seinem höchstwertigen Bit, dem Bit 15 (die Zählweise beginnt immer bei 0), die Information über sein Vorzeichen. Dadurch können in einem Wort Werte von -32768 bis +32767 dargestellt werden, für ein Byte gilt der Bereich von -128 bis +127. Der Wert -1 wird in Bytes durch $FF dargestellt, in Worten durch $FFFF usw.

Bleiben wir bei den positiven Werten. Um die Darstellung eines Bytes im Verhältnis zu seinem Bitmuster einfacher zu gestalten, verwendet man in der Maschinensprache, in der dieses Bitmuster mitunter sehr wichtig ist, nicht mehr das bekannte Dezimalsystem. Üblich sind dagegen das Binär-, das Oktal- und das Sedezimalsystem.

1.2.3 Zahlensysteme

Betrachten wir das Dezimalsystem. Bei diesem System ist die Basis 10, d.h. jede Stelle einer Zahl repräsentiert eine 10er-Potenz. So bedeutet z.B. die Zahl 246 2*10^2+4*10^1+6*10^0. Der Zeichenvorrat besteht hier aus 10 Zeichen, eben 0 bis 9.

Anders ist dies beim Binärsystem. Der Zeichenvorrat dieses Systems ist 2, 0 und 1. Somit ist auch die Basis 2. Die binäre Zahl 1010 bedeutet also:

```
1*2^3+0*2^2+1*2^1+0*2^0=2^3+2^1=8+2=10 (im Dezimalsystem)
```

Binäre Zahlen werden im allgemeinen mit dem %-Zeichen gekennzeichnet. Versuchen Sie also bitte einmal, den dezimalen Wert der Zahl %110011 zu ermitteln!

Na, haben Sie 51 heraus? Stimmt! Die einfachste Methode dieser Umrechnung ist es, bei einer 1 in der Zahl den Stellenwert aufzuaddieren. Der Stellenwert ist bei den ersten 8 Stellen:

Stelle:	8	7	6	5	4	3	2	1
Wert:	128	64	32	16	8	4	2	1

Ähnlich ist es beim Oktalsystem. Dessen Basis ist 8, der Zeichenvorrat 0 bis 7. Die Zahl 31 im Oktalsystem hat somit den Dezimalwert 3*8^1+1*8^0=25. Dieses System ist jedoch nicht so wichtig wie das nächste:

Das Sedezimalsystem, meist Hexadezimalsystem genannt, hat als Basis die 16. Der Zeichenvorrat ist 0-F, also ist 10 dezimal $A, wobei das $-Zeichen auf das Sedezimalsystem hinweist. Dieses System ist neben dem Binärsystem das wichtigste für die Assembler-Programmierung.

Ein Byte, welches ja die Zahlen 0-255 dezimal beinhalten kann, wird im Sedezimalsystem immer zweistellig dargestellt: $00 bis $FF. Ein Wort liegt zwischen $0000 und $FFFF, ein Langwort zwischen $00000000 und $FFFFFFFF.

Die Umrechnung einer Binärzahl in eine Sedezimalzahl, welche man auch als Hexzahl bezeichnet, ist sehr einfach: man teilt einfach die Binärzahl in Vierergruppen auf. Jede dieser Vierergruppen repräsentiert eine Stelle der Sedezimalzahl. Ein Beispiel:

```
Binärzahl:      %110011101111

aufteilen in    %1100 %1110 %1111
ergibt            $C    $E    $F

also:           %110011101111=$CEF
```

Die Umrechnung in der anderen Richtung ist natürlich ebenso leicht:

```
Hexzahl:        $E30D

aufteilen in     $E    $3    $0    $D
ergibt          %1110 %0011 %0000 %1101

also:           $E30D=%1110001100001101
```

Auf diese Art kann übrigens auch mit Oktalzahlen verfahren werden, nur daß man dabei die Binärzahl in Dreierpäckchen aufteilt.

```
Oktalzahl:      7531

aufteilen in     7     5     3     1
ergibt          %111  %101  %011  %001

also     oktal 7531=%111101011001
```

Von der binären Form kann man dies auch wieder in die sedezimale Form bringen:

```
Binärzahl:      %111101011001

aufteilen in    %1111 0101 1001
ergibt            $F   $5   $9

also     oktal 7531 = $F59
```

Um diese Zahl nun in das Dezimalsystem zu bringen, genügt folgende Rechnung:

```
Hexzahl:        $F59

aufteilen in    $F       $5     $9
ergibt          15*16^2 +5*16 +9

also            $F59 = 3929 dezimal
```

Diese Umwandlungen sind manchmal unerläßlich und dennoch lästig. Mit dem SEKA-Assembler kann man sich dies leicht machen, indem man mit '?' den Wert eingibt und den Wert in sedezimaler und dezimaler Schreibweise erhält.

Oft jedoch muß die Umwandlung innerhalb eines Programms erfolgen, wenn man z.B. in einem laufenden Programm eine Zahl eingibt, die dann weiterverarbeitet werden soll. Die eingegebene Zahl liegt dann zunächst als Zeichenfolge vor, die dann ausgewertet und normalerweise in eine Binärzahl, also ein Wort bzw. Langwort im Speicher, umgewandelt werden muß.

Auch andersherum ist es oft nötig, Zahlen umzuwandeln. Will man nämlich einen im Programm errechneten Wert in irgendeinem Zahlensystem ausgeben, so muß der Wert in eine Zeichenfolge umgerechnet werden.

Für diese beiden Probleme werden wir in einem späteren Kapitel Maschinenprogramme entwickeln, welche Sie dann auch evtl. in eigene Programme einsetzen können. Zuerst jedoch müssen wir uns noch einige grundlegende Dinge ansehen, die für das Programmieren unerläßlich sind.

1.3 Aufbau des Amiga

Um die Maschinensprache zu beherrschen, reicht es nicht ganz aus, 'nur' den Prozessor und dessen Befehlssatz zu kennen. Man muß ebenso den Rechner kennen, auf dem man programmiert. Sehen wir uns daher einmal den Amiga von innen an.

1.3.1 Struktur

Der Amiga ist eine sehr fähige Maschine. Dies liegt sehr stark daran, daß außer dem Prozessor selbst noch einige andere Schwerarbeiter in ihm eingebaut sind. Gemeint sind damit die sogenannten Custom-Chips, welche etliche Aufgaben selbständig übernehmen.

Dies sind vor allem drei Chips, welche die poetischen Namen Agnus, Denise und Paula tragen. Agnus, auch als Blitter bezeichnet, betätigt sich hauptamtlich als Speicherverschieber, was

z.B. für schnelle Bildbearbeitung nützlich ist. Denise sorgt dafür, daß wir auf dem Monitor etwas sehen können, und Paula ist für Ein-/Ausgaben zuständig (Diskettenoperationen oder Sound).

Diese Chips werden vom Prozessor über einige Adressen bedient, welche ab $DFF000 liegen und als Hardware-Register bezeichnet werden (siehe auch im entsprechenden Kapitel dieses Buches). Um die teilweise recht komplizierte Bedienung dieser Chips zu erleichtern, sind sowohl im Kickstart als auch auf der Workbench-Diskette einige Bibliotheken enthalten, in denen sich fertige Service-Programme befinden. Diese Programme lassen sich durch mehr oder weniger einfache Programmteile aufrufen und übernehmen dann die Bedienung der entsprechenden Chips.

Verwendet man ausschließlich diese Bibliotheksfunktionen in seinen Programmen, so ist die Programmierung des Amiga in jeder Sprache ähnlich. Egal, ob Assembler, BASIC oder C, die an die Funktionen zu übergebenen Parameter sind die gleichen, nur die Schreibweise ist anders. BASIC ist hier allerdings mit Vorbehalt zu nennen, da einige Befehle dieser Sprache vom BASIC-Interpreter in die entsprechenden Funktionsaufrufe übersetzt werden, weshalb man für die BASIC-Programmierung eigentlich nicht wissen muß, wie der Amiga diese oder jene Funktion ausführt.

Wir bleiben jedoch hier in der untersten Ebene, der Maschinensprache. Die Funktionen, welche die Bibliotheken enthalten, sind ebenfalls Maschinenprogramme und daher mit unseren Programmen eng verwandt. Wir können daher eigentlich auch ganz auf die Verwendung der Bibliotheken verzichten und alles selber machen, was aber so viel Arbeit ist, daß wir dies nicht unbedingt versuchen sollten.

1.3.2 Speicher-Aufteilung

Ausgehend von einem Amiga 1000 kann man zuerst einmal den RAM-Speicher betrachten. Im Normalausbau verfügt dieser Computer über 512 KByte RAM, welcher die Adressen 00000 bis $7FFFF, also 0-524287, belegt. Rüstet man ihn auf ein

Megabyte auf, so liegen die zusätzlichen 512 KByte nicht direkt im Anschluß, sondern von $200000 bis $9FFFFF. Dieser Bereich unterscheidet sich jedoch etwas von dem unteren.

Die Chips, die im Amiga den Prozessor unterstützen, greifen fast alle selbstständig auf den Speicher zu und entlasten so den Prozessor. Es gibt dabei allerdings eine Einschränkung: diese Chips können nur die ersten 512 KByte benutzen. Somit können die Bild- und Ton-Daten, die diesen Chips zugänglich sein sollen, nur in diesem Speicherbereich liegen, weshalb man diesen auch Chip-RAM nennt.

Das Gegenstück zu diesem Chip-RAM ist der restliche RAM-Bereich, welcher, wenn überhaupt eingebaut, über $200000 liegt. Da der Prozessor dort alleine zugreifen kann, ohne von den anderen Chips 'gestört' zu werden, wird dieser Speicher Fast-RAM (Schnell-RAM) genannt.

Hier nun eine Übersicht über die Speicherbelegung im Amiga:

$000000-$07FFFF	Chip-RAM
$080000-$1FFFFF	reserviert
$200000-$9FFFFF	mögliches Fast-RAM
$A00000-$BEFFFF	reserviert
$BFD000-$BFDF00	PIA B (gerade Adressen)
$BFE001-$BFEF01	PIA A (ungerade Adressen)
$C00000-$DFEFFF	reserviert für spätere Erweiterungen
$DFF000-$DFFFFF	Hardware-Register der Custom-Chips
$E00000-$E7FFFF	reserviert
$E80000-$EFFFFF	Adreßbereich des Erweiterungs-Ports
$F00000-$F7FFFF	reserviert
$F80000-$FFFFFF	System-ROM

Wenn wir nun ein Programm in den Speicher laden lassen, so wird dies irgendwohin geladen. Der so belegte Speicherplatz wird dann in einer Belegungsliste gekennzeichnet und gilt somit als gesperrt für andere Anwendungen. Lädt man dann noch ein Programm, was beim Amiga ja ohne weiteres möglich ist, so wird dies wieder woanders hin geladen und belegt weiteren Speicherplatz. Wenn das erste Programm also zusätzlichen Speicherplatz z.B. für Texte braucht, so muß es sich diesen erst ein-

mal reservieren. Andernfalls könnte es sein, daß dieser Speicher und damit die Texte von einem weiteren Programm überschrieben wird.

Interessant dabei ist, daß man nun das erste Programm beenden und somit seinen Speicherplatz wieder freigeben kann. Der Speicher ist dann in freie und belegte Teile unterteilt, welche nicht mehr zusammenhängen. Dennoch kann der Amiga mit diesen Speicherbröckchen noch arbeiten, als ob sie an einem Stück wären! Dies zeigt sich besonders an der dynamischen RAM-Disk, welche immer unter dem Namen RAM: verfügbar ist.

Diese RAM-Disk ist eigentlich ein Phänomen, da sie immer 100% gefüllt ist. Wird ein Programm von dieser RAM-Disk gelöscht, so wird der damit belegte Speicherplatz, egal wo und wie er verteilt ist, wieder vollständig dem System zurückgegeben. Wenn Sie also einen Speicherbereich von z.B. 100 KByte reservieren und füllen, kann es durchaus sein, daß dieser in Wirklichkeit gar nicht zusammenhängend im Speicher liegt. Davon merken Sie als Programmierer jedoch nichts, da er dennoch als zusammenhängend gehandhabt werden kann. Die Differenzen zwischen scheinbarem und physikalischem Speicher werden vom Amiga selbständig korrigiert.

1.3.3 Alles auf einmal: Multitasking

Der Amiga ist schon eine tolle Kiste, kann er doch mehrere Dinge gleichzeitig tun! In dem einen Fenster springt ein rot-weißer Ball hin und her, während man in einem anderen Fenster einen Text bearbeiten kann und gleichzeitig noch eine Uhr in der Ecke vor sich hin tickt.

Das ist jedenfalls der Eindruck, welchen man bekommt, wenn man den Amiga vorgeführt bekommt. Die Sache hat allerdings einen Haken: auch im Amiga ist nur ein Prozessor eingebaut. Dieser Prozessor kann nun wirklich nicht mehrere Dinge gleichzeitig tun.

Der Trick bei der Sache ist lediglich, daß, wenn mehrere Programme gleichzeitig laufen sollen, jedes Programm der Reihe nach ein bißchen bearbeitet wird. Bei obigem Beispiel wird also erst der Ball um einen Bildpunkt weiterbewegt, dann nachgesehen, ob in der Textverarbeitung ein Zeichen eingegeben wurde, und dies ggf. ausgegeben und dann die Uhr ggf. um eine Sekunde weitergestellt. Danach beginnt dies wieder von vorne usw. usw. Das Problem hierbei ist, daß auch der Amiga langsamer wird, wenn er zu viele solche Teilaufgaben zu erledigen hat.

Jeder dieser Jobs, die der Amiga 'gleichzeitig' zu bearbeiten hat, wird Task genannt. Deshalb wird auch die Bearbeitung mehrerer Tasks gleichzeitig 'Multitasking' genannt. Jeder Task, welcher in die Arbeitsfolge des Prozessors eingefügt wird, bekommt dann eine bestimmte Zeit zugeteilt, in der er bei jedem Durchlauf bearbeitet wird. Man kann diese Zeit auch variieren, so daß zeitkritische Programme auch genug bearbeitet werden können.

Wie die Zeiteinteilung des Prozessors nun genau funktioniert, ist für den Programmierer gar nicht so wichtig. Sie können ein Programm ohne Rücksicht darauf schreiben und starten, es wird dann automatisch 'nebenbei' laufen. Dabei ist nur zu beachten, daß Sie das Programm entweder aus dem CLI mit 'run' oder von der Workbench starten müssen. Ein nur mit Namen gestartetes Programm belegt die gesamte Zeit des Prozessors, die es vom CLI bekommen kann, und kehrt erst nach Beendigung seiner Arbeit wieder zum CLI zurück. Starten Sie es jedoch mit 'run', so ist der CLI direkt weiter verfügbar.

Eine weitere Einschränkung für die Möglichkeit des Multitasking gibt es, welche ganz besonders für Assembler-Programme gilt. Abgesehen von der Verwendung zusätzlichen Speichers, welcher unbedingt vorher reserviert werden muß, sollte man auch nicht die Hardware-Register direkt ansprechen. Stattdessen müssen die entsprechenden Funktionen der Bibliotheken verwendet werden. Der Grund dafür ist einfach:

Wenn Sie z.B. den Druckerport als Eingabe definiert haben und Daten einlesen, könnte es ja sein, daß plötzlich irgendein anderer Task meint, jetzt drucken zu müssen. Die Leitungen werden

also auf Ausgang geschaltet und Daten ausgegeben. Danach versucht Ihr Programm auch weiterhin, Daten einzulesen, was ihm dann wohl nicht mehr gelingt!

Dies ist zwar nur ein einfaches Beispiel, aber es verdeutlicht das Problem. In Wirklichkeit gibt es viel schlimmere Effekte, die durch unkontrollierte Mehrfach-Programmierung von einigen Hardware-Registern auftreten können. Wenn Sie also ein Programm dennoch direkt auf die Hardware-Register zugreifen lassen (was einige Vorteile bringen kann), so sorgen Sie besser dafür, daß dieses Programm alleine läuft!

2. Der Prozessor MC68000

Der im Amiga eingebaute Prozessor MC68000 ist ein sehr leistungsfähiger Baustein, welcher auch im ATARI ST oder im Macintosh verwendet wird. Es handelt sich dabei um einen 16/32-Bit-Prozessor, d.h. er kann zwar Daten von 32 Bit verarbeiten, hat aber 'nur' einen 16-Bit breiten Datenbus und einen 24-Bit-Adreßbus. Er kann somit einen Adreßbereich von 2^24=16777216 Byte direkt ansprechen, also 16 MByte.

Der Prozessor im Amiga wird mit etwa 7.1 Megahertz getaktet und ist somit sehr schnell, was für einen so arbeitsintensiven Job wie im Amiga durchaus notwendig ist. Allerdings wird er gerade im Amiga sehr tatkräftig von einigen Peripheriebausteinen unterstützt, was ihn bei vielen Arbeiten sehr gut entlastet. So werden die Graphikausgaben, Tonerzeugung und Ein-/Ausgabe von diesen Bausteinen teilweise übernommen und lassen dem Prozessor somit Zeit, sich um Berechnungen zu kümmern.

2.1 Register

Zusätzlich zum normalen RAM-Speicher hat der Prozessor auch noch einige interne Speicher: die Register. Dies sind 8 Datenregister (D0-D7), 8 Adreßregister (A0-A7), ein Statusregister (SR), zwei Stapelzeiger (USP und SSP) und den Adreßzähler (Programcounter=PC).

Die Datenregister, Adreßregister und der Adreßzähler sind 32 Bit breit, das Statusregister 16 Bit. Die Register werden nicht wie irgendeine Speicherzelle angesprochen, da sie nicht im Speicher, sondern im Prozessor selbst liegen. Wie Sie später bei den Adressierungsarten und im Befehlssatz sehen, gibt es für sie spezielle Befehle.

Die Datenregister werden für Daten aller Art verwendet. Es sind damit Operationen mit Bytes (8 Bit), Worten (16 Bit) und Langworten (32 Bit) möglich, so daß ihre Anwendung sehr flexibel ist.

Die Adreßregister werden allgemein für die Speicherung und Verarbeitung von Adressen verwendet. Sie können so leicht als Zeiger auf Tabellen o. ä. eingesetzt werden. Hierbei sind nur Wort- und Langwort-Operationen möglich.

Eine besondere Rolle spielt das Adreßregister A7. Dieses Register wird immer vom Prozessor als Stapelzeiger (Stackpointer=SP) verwendet und ist somit für den normalen Gebrauch nicht zu empfehlen! Welcher der beiden möglichen Stapel durch diesen SP angezeigt wird, hängt vom momentanen Modus des Prozessors ab; doch dazu später.

Dieser Stapel, auf dessen aktuelle Position der SP zeigt, wird zum Ablegen von internen Daten verwendet, welche nur vorübergehend gerettet werden sollen. Der Stapel funktioniert etwa genauso wie ein Stapel Notizzettel auf Ihrem Schreibtisch: Was zuletzt daraufgelegt wurde, wird auch zuerst heruntergeholt. Diese Art von Stapeln nennt man LIFO (Last In, First Out). Es gibt auch noch die andere Art, den FIFO (First In, First Out)-Stapel, welcher aber vom Prozessor selbst nicht verwendet wird.

Wie man dieses Register und damit den SP manipulieren bzw. wie man mit dem Stapel arbeiten kann, finden Sie u.a. im nächsten Kapitel. Doch nun zurück zu den Registern.

Das Statusregister SR spielt für die Maschinenprogrammierung eine ganz besondere Rolle. In diesem Wort sind in 10 Bits einige wichtige Informationen enthalten, die Aufschluß über den Status des Prozessors geben. Man unterteilt dieses Wort in das untere Byte, das User-Byte, und das obere, das System-Byte. Die Bits, welchen eine Bedeutung zukommt, werden als Flags bezeichnet. Das heißt, daß hier ein bestimmter Zustand angezeigt wird, wenn das Bit gesetzt ist.

Im User-Byte sind 5 Flags enthalten, welche folgende Bedeutungen haben:

Bit	Name	Bedeutung
0	*(C, Carry)*	Übertragsbit. Wird bei Verknüpfungen, Berechnungen und Verschiebe-Befehlen ggf. verändert
1	*(V, Overflow)*	Ist ähnlich wie Carry, zeigt jedoch eine Änderung des Vorzeichens bzw. einen Übertrag von Bit 6 in Bit 7 an
2	*(Z, Zero)*	Wird gesetzt, wenn das Ergebnis einer Operation Null ist
3	*(N, Negativ)*	Wird gesetzt, wenn das Ergebnis einer Operation negativ ist (höchstes Bit gesetzt)
4	*(X, Extended)*	Wird bei arithmetischen Operationen ggf. gesetzt wie das Carry
5-7		Unbenutzt

Im Systembyte sind ebenfalls 5 Bits von Bedeutung:

Bit	Name	Bedeutung
8 *9* *10*	*I0* *I1* *I2*	Interrupt-Maske. Hiermit wird der Interrupt-Level von 0 bis 7 eingestellt, wobei 0 die niedrigste und 7 die höchste Priorität darstellt, in dem Interrupts erlaubt werden
11		Unbenutzt
12		Unbenutzt
13	*(S, Supervisor)*	In diesem Bit ist der aktuelle Modus enthalten, in dem sich der Prozessor befindet (0=User-, 1=Supervisor-Modus).
14		Unbenutzt
15	*(T, Trace)*	Ist dieses Bit gesetzt, so befindet sich der Prozessor im Einzelschritt-Modus.

Hier eine Übersicht über das gesamte Statuswort:

Bit :	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name:	T	-	S	-	-	I2	I1	I0	-	-	-	X	N	Z	V	C

Hier sind einige Begriffe wie Modus oder Interrupt aufgetaucht, von denen Sie sich nicht verwirren lassen sollten. Diese Dinge werden später im Kapitel über die Betriebszustände des Prozessors behandelt.

2.2 Speicher-Adressierung

Der Prozessor verfügt in einem normal ausgebauten Amiga 500 oder 1000 über 512 KByte RAM, der Amiga 2000 hat sogar 1 MByte, in denen er nach Belieben Daten ablegen kann. Doch wie kommt er an diese Daten heran? Wir wollen uns diese Problematik nun einmal genau ansehen.

Programmiert man z.B. in BASIC, braucht man sich zunächst einmal über die Aufteilung des Speichers keine Gedanken zu machen. Man schreibt einfach MERKER%=1, und der Wert ist gespeichert.

Es gibt nun zwei Wege, diesen Vorgang in Assembler zu realisieren:

1) man legt diesen Wert in eines der Daten- oder Adreß-Register oder

2) man schreibt ihn in irgendeine Speicherzelle.

Um diese beiden Wege einmal zu zeigen, greifen wir ein wenig vor und betrachten den ersten Maschinen-Befehl, welcher auch wohl der häufigste ist: MOVE. Mit diesem Befehl wird, wie der Name schon sagt, ein Wert verschoben. Diesem Befehl werden noch zwei Angaben beigegeben: Quelle und Ziel.

Sehen wir uns nun das obige Beispiel an, wenn wir den MOVE-Befehl anwenden:

zu 1) `MOVE #1,D0`

legt den Wert 1 im Datenregister D0 ab. Wie Sie sehen, wird immer erst der Quell- und dann der Ziel-Operand angegeben. Somit ist die Anweisung MOVE D0,#1 unmöglich.

zu 2) `MOVE #1,$1000`

legt den bekannten Wert 1 in die Speicherzelle $1000. Diese Adresse ist willkürlich gewählt. Meistens tauchen in mit Assemblern geschriebenen Programmen überhaupt keine Adressen in dieser Form auf, da man dort sogenannte Labels einsetzt, die auf eine bestimmte Stelle im Programm verweisen. So schreibt man, will man die 1 in eine Speicherzelle im Programm schreiben, ungefähr folgendes:

```
        ...
        MOVE #1,MERKER

        ...
MERKER:   DC.W 1
```

Dies sind zwei Stellen eines Programms. Die obere Zeile führt den üblichen MOVE-Befehl aus, wobei die Zieladresse 'MERKER' ist. Dieser Label taucht dann später wieder auf und markiert die Speicherzelle, in die der Wert geschrieben werden soll.

Die Anweisung DC.W 1 ist ein sogenannter Pseudo-Op. Dies bedeutet, daß es sich dabei nicht um einem Maschinen-Befehl für den Prozessor, sondern um eine Anweisung an den Assembler handelt. DC steht dabei für 'declare', frei übersetzt mit 'reserviere', der Zusatz .W bedeutet, daß es sich in diesem Fall um ein Datenwort (.W wie Wort) handelt. Weitere Möglichkeiten sind hier .B für Byte (8 Bit) und .L für Langwort (32 Bit).

Dieser Zusatz (.B, .W oder .L) kann auch bei den meisten Maschinenbefehlen verwendet werden. Läßt man ihn weg, so setzt

der Assembler normalerweise Wortbreite voraus und assembliert es als .W. Will man ein Langwort verwenden, so schreibt man z.B. MOVE.L #$12345678,D0, ein Byte wird mit MOVE.B #$12,D0 geschrieben. Hierbei ist eines unbedingt zu beachten:

Hinweis: Wird mit Wort- oder Langwort-Operationen auf den Speicher zugegriffen, so muß diese Adresse unbedingt gerade sein (Endziffer 0,2,4,6,8,A,C,E)!

Dies kann bei Verwendung eines Assemblers durch die Anweisung 'EVEN' erreicht werden. Nötig wird dies immer, wenn ungefähr folgendes geschrieben wird:

```
        ...
WERT1:    DC.B 1
WERT2:    DC.W 1
```

Liegt die Speicherzelle WERT1 an einer geraden Adresse, so liegt WERT2 zwangsläufig an einer ungeraden. Schiebt man aber hier ein EVEN ein, so wird hier ggf. ein Füllbyte (0) vom Assembler eingefügt, und die Adresse ist wieder gerade.

```
        ...
WERT1:    DC.B 1
EVEN
WERT2:    DC.W 1
```

Zurück zu unserem Beispiel mit den Adressierungsarten. Die oben angegebenen Variationen sind also vergleichbar mit dem BASIC-Befehl MERKER%=1, wobei das %-Zeichen auf einen Integer-Wert hinweist.

Nun gehen wir einen Schritt weiter und 'übersetzen' einmal MERKER%=WERT%. Ich nehme an, Sie wissen schon die Lösung, nicht wahr? Dann vergleichen Sie Ihre Idee mit folgendem:

```
        MOVE WERT,MERKER
        ...
        ...
MERKER:   DC.W 1
WERT:     DC.W 1
```

Hier wird anstelle des direkten Wertes 1 lediglich der Inhalt der Speicherzelle an der Adresse 'WERT' in die Adresse MERKER ge'move't.

Wir haben mit Hilfe dieser kleinen Beispiele bereits vier Adressierungsarten kennengelernt, in denen der Prozessor auf den Speicher zugreifen kann. Die erste ist gekennzeichnet durch das #-Zeichen und stellt einen direkten Wert dar. Diese Adressierungsart heißt daher 'unmittelbare Adressierung'. Sie ist natürlich nur für den Quell-Operanden erlaubt!

Eine weitere Art, bei der eine direkte Adresse (hier 'MERKER' und 'WERT') angegeben ist, heißt einfach 'absolute Adressierung'. Sie ist sowohl für den Quell- als auch für den Ziel-Operanden möglich, wie wir gesehen haben.

Diese wird in zwei unterschiedliche Adressierungsarten unterschieden, wovon der Programmierer normalerweise nichts merkt. Je nachdem, ob die absolute Adresse größer oder kleiner als $FFFF ist, also ein Langwort benötigt oder nicht, heißt es nämlich 'absolut lang' bei Adressen >$FFFF oder andernfalls 'absolut kurz'. Die Unterscheidung, um welche Adressierungsart es sich jeweils handelt, trifft aber normalerweise der Assembler, so daß Sie sich nur allgemein die absolute Adressierung merken brauchen.

Die vierte bisher aufgetauchte Adressierungsart heißt 'Datenregister direkt'. Sie wurde als erstes vorgestellt (MOVE #1,D0) in Verbindung mit der unmittelbaren Adressierung und bedeutet nichts anderes, als daß ein Datenregister (hier D0) direkt verwendet wurde.

Nun gibt es aber beim 68000-Prozessor nicht nur diese vier Adressierungsarten, sondern deren 12. Eine weitere Art heißt z.B. 'Adreßregister direkt' und entspricht der 'Datenregister

direkt'-Adressierung, nur daß hier ein Adreßregister verwendet wird. So kann man mit MOVE.L #MERKER,A0 die Adresse des Merkers direkt ins Adreßregister A0 bringen.

Das wären also schon 5 Adressierungsarten, mit denen allein schon einiges anzufangen ist. Doch nun wird es ein wenig komplizierter, aber auch interessanter.

Nehmen wir wieder ein Beispiel aus der Programmiersprache BASIC, und zwar folgendes Programmstückchen:

```
10  A=1000
20  POKE A,1
```

Hier wird zunächst der Variablen A der direkte Wert 1000 zugeordnet. Das können wir auch in Assembler: MOVE.L #1000,A0. Hier wird der absolute Wert 1000 als Langwort ins Adreßregister A0 geschrieben.

Doch nun zur Zeile 20. Hier wird nicht der Variablen A selbst ein Wert zugeordnet, sondern der Speicherzelle, deren Adresse in A steht. Es handelt sich dabei also um einen indirekten Zugriff, der auch in Assembler leicht zu realisieren ist:

```
MOVE.L  #1000,A0    ;Adresse in A0 bringen
MOVE    #1,(A0)     ;und die 1 in diese Adresse schreiben
```

Durch die Klammer, die um A0 steht, wird die Adressierungsart vorgegeben, welche 'Adreßregister indirekt' heißt. Diese Adressierungsart funktioniert somit nur mit Adreßregistern, ein 'Datenregister indirekt' gibt es nicht!

Es gibt nun auch einige Variationen dieser Adressierungsart. So kann man in der Assembler-Anweisung einen Distanzwert angeben, welcher vor dem Zugriff auf die im Adreßregister enthaltene Adresse zu dieser addiert wird. MOVE #1,4(A0) in obiges Beispiel eingesetzt, schreibt also die 1 in die Speicherzelle 1000+4=1004. Dieser Distanzwert darf 16 Bit breit sein und wird vorzeichenrichtig behandelt. Somit kann er einen Wert von -32768 bis +32767 annehmen. Diese Adressierungsart heißt 'Adreßregister indirekt mit 16-Bit-Distanzwert'.

Hierzu gibt es eine Erweiterung: 'Adreßregister indirekt mit 8-Bit-Distanzwert', bei der jedoch nicht nur der Distanzwert auf 8 Bit begrenzt ist, sondern auch noch ein weiteres Register auftaucht. Dieses zweite Register zählt dabei ebenfalls als Distanzwert, nur daß dieser natürlich auch variabel ist.

Verdeutlichen wir dies mit einem Beispiel. Angenommen, wir haben eine Datenliste im Programm vorliegen, die folgendermaßen aufgebaut ist:

```
        ...
LISTE:    DC.W 2          ;Anzahl der Einträge-1
          DC.W 1,2,3      ;Listen-Elemente
```

Wir laden nun mittels MOVE.L #LISTE,A0 die Adresse dieser Liste in das Adreßregister A0. Nun können wir mit MOVE (A0),D0 die Anzahl der Listenelemente in das Datenregister holen. Um nun auf das letzte Element der Liste zuzugreifen, brauchen wir nur einen Befehl. Dies alles sieht dann so aus:

```
        CLR.L  D0           ;D0 komplett löschen
        MOVE.L #LISTE,A0    ;Adresse der Liste in A0
        MOVE   (A0),D0      ;Anzahl der Elemente-1 in D0
        MOVE   1(A0,D0),D1  ;und letztes Element in D1
            ...
LISTE:    DC.W 2            ;Anzahl der Einträge-1
          DC.W 1,2,3        ;Listen-Elemente
```

Dieser letzte Befehl greift also auf das Byte zu, welches in 1+A0+D0 liegt, also der Liste+1, wo die Daten beginnen, plus dem Inhalt von D0, in diesem Fall 2.

Mit dieser Adressierungsart kann man sichtlich sehr viel anfangen. Sie eignet sich hervorragend für die Verarbeitung von Listen und Tabellen, wie das Beispiel zeigt. Benötigt man keinen Distanzwert, so muß man eine 0 einsetzen, was jedoch so mancher Assembler von alleine erledigt, wenn man nur z.B. MOVE (A0,D0) eingibt.

Für die beiden zuletzt vorgestellten Adressierungsarten gibt es eine weitere Variante, welche eine Besonderheit aufweist. Es

wird dabei nicht irgendein Adreßregister verwendet, sondern der Programmzähler (PC). Diese Adressierungsart findet vor allem dann Verwendung, wenn man sein Programm so gestalten möchte, daß es ohne Änderung in jedem beliebigen Speicherbereich läuft. So haben (in den 15-Bit-Grenzen) folgende beiden Anweisungen denselben Effekt:

```
MOVE  MERKER,D0
```

und

```
MOVE  MERKER(PC),D0
```

Diese Schreibweise ist eigentlich ungenau, da in der oberen Anweisung mit MERKER die Adresse des Merkers, in der unteren der Abstand zwischen dieser Anweisung und dem Merker gemeint ist. Da es jedoch umständlich wäre, jedesmal den Abstand zu berechnen, nimmt ein Assembler das dennoch so an und berechnet den wirklichen Wert selbst.

Doch betrachten wir nun den Unterschied zwischen der oberen und der unteren Anweisung. Sie bewirken zwar im Programm dasselbe, werden jedoch vom Assembler völlig unterschiedlich übersetzt. Nehmen wir an, das Programm stünde ab Adresse $1000 und der Merker in Adresse $1100. Der generierte Programmcode ist dann folgender:

```
$001000   30 39 00 00 11 00   MOVE  MERKER,D1
```

bzw.

```
$001000   30 3A 00 FE         MOVE  MERKER(PC),D1
```

Wie Sie sehen, ist der generierte Code in der zweiten Version um zwei Byte kürzer als der der ersten Version. Zusätzlich entsteht der Effekt, daß, wenn Sie diesen Code z.B. in den Speicherbereich ab $2000 verlegen, die erste Version immer noch auf $1100 zugreift, während mit der PC-indirekten Adressierung richtig auf $2100 zugreift: das Programm ist verschiebbar.

Dies ist also die Adressierungsart 'Programmzähler indirekt mit 16-Bit-Distanzwert'. Daneben gibt es auch noch, wie erwähnt,

die Art 'Programmzähler indirekt mit 8-Bit-Distanzwert', die zusätzlich ein weiteres Register als Distanzangabe, auch Offset genannt, zuläßt.

Haben Sie mitgezählt? Richtig, es fehlen noch zwei Adressierungsarten. Auch diese beiden bauen auf die indirekte Adressierung auf. Man hat nämlich die Möglichkeit, beim Speicherzugriff mit 'Adreßregister indirekt' das Adreßregister automatisch um eins zu erhöhen oder zu erniedrigen.

Zum automatischen Erhöhen dient die Adressierungsart 'Adreßregister indirekt mit Postinkrement', welche das Adreßregister NACH dem Zugriff um die Anzahl der verwendeten Bytes erhöht. Schreibt man also z.B.

```
MOVE.L  #1000,A0
MOVE.B  #1,(A0)+
```

so wird die 1 in die Adresse 1000 geschrieben und dann A0 um eins erhöht. Ein solcher Befehl ist sehr nützlich, wenn man einen Speicherbereich mit einem bestimmten Wert ausfüllen will (z.B. den Bildschirmspeicher löschen). Man kann dafür diesen Befehl in eine Schleife legen - wie das geht, sehen wir später.

Das Gegenstück zum Postinkrement ist die Adressierungsart 'Adreßregister indirekt mit Predekrement'. Hierbei wird das angegebene Adreßregister VOR dem Zugriff um eins erniedrigt. Somit bewirkt

```
MOVE.L  #1000,A0
MOVE.B  #1,-(A0)
```

das Schreiben der 1 in die Adresse 999, da zuerst der Inhalt von A0 erniedrigt und dann die 1 geschrieben wird.

Diese beiden Adressierungsarten werden für die Verwaltung des Stapelzeigers (SP) verwendet. Da der Stapel, oft auch Stack genannt, von oben nach unten gefüllt wird, schreibt man zum Ablegen eines Wortes z.B. aus D0 auf den Stapel:

```
MOVE.B  D0,-(SP)
```

und zum Herunternehmen vom Stapel wieder in D0:

```
MOVE.B  (SP)+,D0
```

Dadurch zeigt der Stapelzeiger immer auf das zuletzt abgelegte Byte auf dem Stapel. Auch hier müssen Sie sehr aufpassen, daß der Wort- oder Langwortzugriff auf den Stapel auf eine gerade Adresse erfolgt. Wenn Sie also ein Byte auf den Stapel bringen, tun Sie das entweder gleich mit Wortbreite oder achten Sie darauf, daß mit dem abgelegten Byte kein JSR oder BSR erfolgt. Diese Befehle legen nämlich die Adresse, wo sie herkommen, auf dem Stapel ab, und zwar als Langwort!

Übrigens wird bei obiger Schreibweise immer anstelle von SP das Adreßregister A7 im Programmcode eingesetzt, da dieses Adreßregister immer als SP verwendet wird. Sie können daher getrost auch A7 statt SP schreiben - das Programm ist dasselbe. Dennoch empfehle ich Ihnen die Verwendung von SP, da dies übersichtlicher wird. Oft nämlich legt man sich in Maschinenprogrammen eigene Stapelbereiche an, auf die ebenfalls so zugegriffen wird. Dann wird die deutliche Unterscheidung zwischen Prozessorstapel und eigenem Stapel sehr wichtig!

Dies waren also die 12 Adressierungsarten des MC68000-Prozessors. Hier noch einmal alle auf einen Blick:

Nr.	Name	Schreibweise
1	Datenregister direkt	Dn
2	Adreßregister direkt	An
3	Adreßregister indirekt	(An)
4	Adreßregister indirekt mit Postinkrement	(An)+
5	Adreßregister indirekt mit Predekrement	-(An)
6	Adreßregister indirekt mit 16-Bit-Distanzwert	d16(An)
7	Adreßregister indirekt mit 8-Bit-Distanzwert	d8(An,Rn)
8	Absolut kurz	xxxx.W
9	Absolut lang	xxxxxxxx.L

10	Unmittelbar	#'Daten'
11	Programmzähler indirekt mit 16-Bit-Distanzwert	d16(PC)
12	Programmzähler indirekt mit 8-Bit-Distanzwert	d8(PC,Rn)

Die hier verwendeten Kürzel bedeuten:

An	Adreßregister A0-A7
Dn	Datenregister D0-D7
d16	16-Bit-Wert
d8	8-Bit-Wert
Rn	Register D0-D7, A0-A7
'Daten'	ein bis zu 32-Bit breiter Wert, je nach verwendeter Datenbreite (.B, .W oder .L)

Dies sind also alle Adressierungsarten, über die der MC68000-Prozessor verfügt. Der große Bruder dieses Prozessors, der 32-Bit-Prozessor MC68020, besitzt noch 6 weitere, auf die wir jedoch hier nicht eingehen wollen.

Schauen wir uns nun einmal an, unter welchen Bedingungen der Prozessor arbeiten kann.

2.3 Betriebszustände

Wir haben im Kapitel über Register bereits das Statusregister (SR) kennengelernt. Die einzelnen Bits dieses Registers geben den Betriebszustand des Prozessors an, der momentan vorliegt. Dabei war eine Unterscheidung zwischen dem Systembyte (Bits 8-15) und dem Userbyte (Bits 0-7) getroffen worden. Beginnen wir mit dem Systembyte und seinen Auswirkungen auf die Arbeit des Prozessors.

2.3.1 User- und Supervisor-Modus

Ist es nicht eigenartig, daß der Prozessor Sie als User (Benutzer) oder Supervisor (Überwacher) einstuft? Diese beiden Betriebszustände sind nämlich möglich, wobei normalerweise der User-Modus vorliegt. In diesem Modus ist es Ihnen nicht erlaubt, verschiedene Maschinenbefehle zu erteilen, und das in Ihrem eigenen Rechner!

Doch keine Panik, diese Beschränkung ist nicht nur sinnvoll, sondern auch umgehbar. Das Amiga-Betriebssystem enthält nämlich eine Funktion, mit der sich der Prozessor in den Supervisor-Modus umschalten läßt. Ohne diese Funktion ist es allerdings sehr umständlich, ja fast unmöglich, diese Umschaltung selbst vorzunehmen.

Der Modus wird durch das Bit 13 des Statusregisters bestimmt. Im Normalfall ist dieses Bit gelöscht (0) und der Prozessor ist im User-Modus. Nun gibt es die Möglichkeit, direkt in das Statusregister zu schreiben, doch ist dieser Maschinen-Befehl privilegiert und somit nur aus dem Supervisor-Modus anwendbar. Diesen Befehl könnte man daher nur zum Umschalten vom Supervisor- in den User-Modus anwenden, indem man z.B. mit AND #$DFFF,SR das Supervisorbit löscht. Sie sehen, die Umschaltung sollten wir lieber dem Betriebssystem überlassen!

Was unterscheidet nun die beiden Modi bezüglich ihrer Anwendung?

Der erste Unterschied wurde bereits erwähnt: einige Maschinenbefehle, wie z.B. MOVE xx,SR, sind privilegiert und können somit nur im Supervisor-Modus ausgeführt werden. Der Versuch, dies im User-Modus zu tun, wird mit einer Unterbrechung des Programms durch eine sogenannte Exception, eine Ausnahme, bestraft. Diese Exceptions werden etwas später betrachtet, sie bieten die einzige Möglichkeit, in den Supervisor-Modus umzuschalten.

Ein weiterer Unterschied liegt im verwendeten Stack-Bereich. Es wird zwar weiterhin A7 als Stapelzeiger (SP) verwendet, jedoch

wird ein anderer Speicherbereich als Stapel benutzt. Beim Umschalten von einem auf einen anderen Modus wird der SP daher immer verändert. Men unterscheidet dann zwischen dem User-SP (USP) und dem Supervisor-SP (SSP).

Auch Speicherzugriffe sind evtl. modusabhängig. Der Prozessor gibt bei einem solchen Zugriff Signale an die Peripherie-Bausteine weiter, die diesen den Modus des Prozessors signalisieren. Dadurch können in einem 68000-Computer verschiedene Speicherbereiche privilegiert werden, so daß man als User nicht darauf zugreifen kann.

Im Supervisor-Modus ist es also möglich, alle Befehle auszuführen und auf alle Speicherbereiche zuzugreifen. Aus diesem Grund laufen Betriebssysteme meist im Supervisor-Modus. Erreicht wird dies durch die Verwendung von Exceptions.

2.3.2 Exceptions

Exceptions, zu deutsch Ausnahmen, treten aus verschiedenen Gründen bzw. Anlässen auf. Tritt ein solcher Anlaß auf, so wird je nach Art dieses Ereignisses in eine bestimmte Routine verzweigt. Vorher jedoch werden einige Schritte vorgenommen:

1) das Statusregister wird gerettet
2) das S-Bit im SR wird gesetzt (Supervisor-Modus) und das T-Bit gelöscht (kein Trace)
3) der Programmzähler und der User-SP werden gerettet
4) der Exception-Vektor, welcher auf die auszuführende Exception-Routine zeigt, wird geholt
5) die Routine wird ausgeführt

Die erwähnten Vektoren, in denen die Adressen der verschiedenen Routinen zur Bearbeitung des Ereignisses stehen, liegen ganz am Anfang des Speichers. Hier eine Übersicht über die Vektoren und ihre Adressen:

Nummer	Adresse	Verwendung bei
0	$000	RESET: Anfangs-SSP
1	$004	RESET: Anfangs-PC
2	$008	Bus Error
3	$00C	Address Error
4	$010	Illegaler Befehl
5	$014	Division durch Null
6	$018	CHK-Befehl
7	$01C	TRAPV-Befehl
8	$020	Privilegsverletzung
9	$024	Trace
10	$028	Axxx-Befehlsemulation
11	$02C	Fxxx-Befehlsemulation
	$030-$038	Reserviert
15	$03C	Uninitialisierter Interrupt
	$040-$05F	Reserviert
24	$060	Unberechtigter Interrupt
25-31	$064-$083	Level 1-7 Interrupt
32-47	$080-$0BF	TRAP-Befehle
	$0C0-$0FF	Reserviert
64-255	$100-$3FF	User-Interrupt-Vektoren

Die einzelnen Begriffe aus obiger Tabelle bedürfen wohl einer etwas näheren Betrachtung. Gehen wir sie also einmal der Reihe nach durch.

RESET: Anfangs-SSP

Dies ist wohl noch einfach zu verstehen: beim Reset wird das hier stehende Langwort als Stackpointer für den Supervisor-Modus (SSP) übernommen. Somit kann man den Stack für die RESET-Routine festlegen.

RESET: Anfangs-PC

Ebenfalls bei RESET wird dieser Wert als Programmzähler übernommen, d.h. die RESET-Routine wird an der hier enthaltenen Adresse gestartet.

Bus Error

Diese Exception wird durch die Hardware bzw. einen Coprozessor ausgelöst, wenn z.B. auf einen reservierten oder nicht existierenden Speicherbereich zugegriffen wird.

Address Error

Dieser Fehler tritt dann auf, wenn mit einem Wort- oder Langwort-Zugriff auf eine ungerade Speicheradresse zugegriffen wird.

Illegaler Befehl

Da alle Befehle des MC68000 aus einem Wort bestehen, können somit 65536 'Befehle' formuliert werden. Da der Prozessor aber natürlich nicht so viele Befehle kennt, sind einige Worte als Befehle ungültig bzw. illegal. Tritt ein solcher Befehl auf, so wird diese Exception ausgelöst.

Division durch Null

Da der Prozessor über einen Divisionsbefehl verfügt und die Division durch Null mathematisch nicht definiert und damit nicht erlaubt ist, tritt bei einer solchen Operation diese Exception auf.

CHK-Befehl

Diese Exception tritt nur ggf. beim CHK-Befehl auf. Dieser Befehl testet ein Datenregister, ob dessen Inhalt sich innerhalb bestimmter Grenzen bewegt. Ist dies nicht der Fall, so wird die Exception ausgelöst.

TRAPV-Befehl

Wird der TRAPV-Befehl ausgeführt, so wird diese Exception ausgelöst, wenn das V-Bit im Statuswort (Bit 1) gesetzt ist.

Privilegsverletzung

Wird im User-Modus ein privilegierter Befehl aufgerufen, so löst dies diese Exception aus.

Trace

Ist im Statuswort das Trace-Bit (Bit 15) gesetzt, so wird diese Exception nach jedem ausgeführten Befehl ausgelöst. Mit dieser Methode läßt sich leicht eine Einzelschritt-Abarbeitung von Maschinenprogrammen realisieren.

Axxx-Befehlsemulation
Fxxx-Befehlsemulation

Diese beiden Vektoren sind für einen interessanten Trick anwendbar. Wird nämlich ein Befehl aufgerufen, der mit $A bzw. mit $F beginnt (z.B. $A010 oder $F200), so wird zu der mit dem entsprechenden Vektor gezeigten Routine gesprungen. In diesen Routinen können nun je nach dem restlichen Befehlswort Funktionen ausgelöst werden, wodurch quasi Befehlserweiterungen für den Prozessor geschrieben werden können!

Reserviert

Diese Vektoren werden nicht verwendet.

Uninitialisierter Interrupt

Diese Exception wird ausgelöst, wenn ein Interrupt von einem Peripherie-Baustein kommt, der nicht initialisiert wurde.

Unberechtigter Interrupt

Dies tritt auf, wenn während der Interrupt-Bestätigung des auslösenden Bausteins ein BUS-Fehler auftritt. In diesem Fall handelt es sich bei dem ausgelösten Interrupt wahrscheinlich nur um eine Störung.

Level 1-7 Interrupt

Diese 7 Vektoren zeigen auf die Interrupt-Routinen der entsprechenden Prioritäten (Levels). Ist der im Statuswort enthaltene Level höher als derjenige des aufgetretenen Interrupts, so wird dieser ignoriert.

TRAP-Befehle

Diese 16 Vektoren werden verwendet, wenn ein entsprechender TRAP-Befehl auftritt. Diese TRAP- Befehle sind daher von TRAP #0 bis TRAP #15 möglich.

User-Interrupt-Vektoren

Diese Vektoren werden für Interrupts verwendet, die von einigen Peripherie-Bausteinen ausgelöst werden, welche eine eigene Vektornummer generieren.

Wir wollen aber an dieser Stelle nicht tiefer in die Geheimnisse und Möglichkeiten der Exception-Behandlung eingehen, damit der Rahmen dieses Buches nicht gesprengt wird. Nur soviel ist noch zu sagen: die Exception-Routinen werden mit dem Befehl RTE (Return from Exception) beendet, wodurch der alte Zustand wiederhergestellt und im User-Programm weitergearbeitet wird.

2.3.3 Interrupts

Ähnlich wie die Exceptions werden die Interrupts bearbeitet. Dies sind Programmunterbrechungen, die von der Hardware, also einem Peripherie-Baustein bzw. einer externen Schaltung ausgelöst werden.

Im Statusregister liegt in den Bits 8-10 ein Wert zwischen 0 und 7, welcher den sogenannten Interrupt-Level darstellt. Es gibt nämlich 8 verschiedene Interrupt-Möglichkeiten, die jeweils eine bestimmte Priorität besitzen. Ist diese Priorität höher als der im Statusregister enthaltene Wert, so wird dieser Interrupt ausgeführt, andernfalls wird er ignoriert.

Tritt ein gültiger Interrupt auf, so wird zu der entsprechenden Routine verzweigt, deren Adresse in der Exception-Vektor-Tabelle (s.o.) enthalten ist.

Solche Programmunterbrechungen sind sehr wichtig, wenn man ein Programm mit der angeschlossenen Hardware synchronisieren

will. So kann z.B. eine Schaltung, die dem Rechner Daten übermitteln will, das Anliegen eines gültigen Wertes mit einem Interrupt signalisieren; die Interrupt-Routine braucht nur den anliegenden Wert direkt zu übernehmen. Diese Methode wird auch bei der Bedienung der seriellen Schnittstelle (RS232) verwendet.

Auf die Anwendung der Interrupts wird später noch eingegangen. Hier sei nur noch erwähnt, daß eine Interrupt-Routine wie auch eine Exception-Routine mit dem RTE-Befehl abgeschlossen wird.

Bevor wir als nächstes das User-Byte des Statusregisters betrachten, noch ein paar Bemerkungen zum Trace-Bit. Wenn dieses Bit gesetzt ist, werden nach wie vor die Exceptions normal ausgeführt. Im normalen Betrieb jedoch, unabhängig vom aktuellen Modus, wird nach jedem ausgeführten Maschinenbefehl eine Exception ausgelöst (Nr. 9, Trace), in welcher dann die Register ect. ausgewertet werden können. Diese Methode wird in Debugger-Programmen wie dem K-SEKA angewandt.

2.3.4 Bedingungscodes

Wenn man ein Programm in irgendeiner Programmiersprache schreibt, taucht immer wieder die Notwendigkeit einer bedingten Operation auf. So z.B. in einem BASIC-Programm:

```
IF D1=2 THEN D2=0
```

Um dies in einem Maschinenprogramm zu realisieren, muß man natürlich zuerst den Vergleich durchführen:

```
CMP  #2,D1
```

CMP bedeutet Compare und bewirkt den Vergleich der gegebenen Operanden, hier D1 mit dem Wert 2. Doch wie kann man das Ergebnis dieser Operation auswerten?

Für diesen Zweck gibt es die sogenannten Bedingungscodes (CC=Condition Code), welche in den Verzweigungsbefehlen der Maschinensprache existieren. Verzweigen heißt in der 'Computersprache' auf Englisch 'to branch', weshalb diese Verzweigungsbefehle Branch heißen. Doch muß dabei natürlich angegeben werden, wohin und vor allem wann verzweigt werden soll.

Der einfachste Fall ist die unbedingte Verzweigung. Der entsprechende Befehl lautet 'BRA Adresse', was für Branch Always steht. Dieser Befehl hilft uns natürlich hier nicht weiter.

Um das Ergebnis einer Operation, hier der Vergleichsoperation CMP, für die Auswertung festzuhalten, sind im Statusregister einige Bits vorgesehen. Betrachten wir zunächst Bit 2, das Zero-Flag. Dieses Bit wird genau dann gesetzt, wenn das Ergebnis der vorangegangenen Operation Null war.

Um den Zusammenhang zwischen dem CMP und dem Z-Flag zu klären, muß zuerst die Funktion des CMP-Befehls betrachtet werden. Dieser Befehl führt nämlich in Wirklichkeit eine Subtraktion des Quell- vom Zieloperanden durch. Im Beispiel wird also vom Inhalt des Datenregisters D1 der Wert 2 subtrahiert. Das Ergebnis wird zwar verworfen, vorher werden aber die entsprechenden Flags gesetzt.

War in unserem Beispiel im Register D1 ebenfalls der Wert 2 enthalten, so ist das Ergebnis der Subtraktion 0. Dadurch wird das Z-Flag gesetzt, welches nun durch einen geeigneten Branch-Befehl ausgewertet werden kann. Das Beispiel sähe dann also so aus:

```
        ...
        CMP   #2,D1      ;Vergleich bzw. Subtraktion
        BNE   UNGLEICH   ;Verzweigen, wenn ungleich (Z gelöscht)
        MOVE  #0,D2      ;sonst D2=0 ausführen
UNGLEICH:
        ...              ;hierhin wird ggf. verzweigt
```

NE steht für Branch if Not Equal, was soviel wie 'Verzweige, wenn nicht gleich' und nach einem CMP-Befehl eine klare

Funktion bedeutet. In Wirklichkeit verzweigt BNE, wenn das Z-Flag gelöscht (=0) ist. Das Gegenstück hierfür wäre der BEQ-Befehl, Branch if Equal (Z=1).

Hier nun eine Auflistung aller Bedingungscodes, die mit einem Bcc (cc = Condition Code) eine bedingte Verzweigung ermöglichen:

cc	Bedingung	Bits
T	True, immer, entspricht BRA	
F	False, nie, verzweigt nie	
HI	higher, höher als	C' * Z'
LS	lower or same, kleiner gleich	C + Z
CC, HS	Carry clear, higher or same	C'
CS, LO	Carry set, lower	C
NE	not equal, ungleich	Z'
EQ	equal, gleich	Z
VC	Overflow clear, kein Überlauf	V'
VS	Overflow set, Überlauf	V
PL	plus, positiv	N'
MI	minus, negativ	N
GE	greater or equal, größer gleich	N*V + N'*V'
LT	less than, weniger als	N*V' + N'*V
GT	greater than, größer als	N*V*Z'+N'*V'*Z'
LE	less or equal, weniger oder gleich	Z + N*V' + N'*V

*=logisch UND, +=logisch ODER, '=logisch NICHT

Hier einige Beispiele dafür, wie diese große Anzahl von Bedingungen verwendet werden können:

```
CMP #2,D1
BLS KLEINER_GLEICH
```

Verzweigt, wenn der Inhalt von D1 <= 2 ist (D1 = 0, 1 oder 2). Der ähnlich klingende Befehl BLE dagegen verzweigt in diesem Beispiel außerdem noch, wenn D1 negativ ist. Dies sieht man daran, daß bei der Auswertung der Bedingung das V-Bit verwendet wird (s. o.), welches bei Vorzeichenwechsel während der

Operation reagiert und mit dem N-Bit verglichen wird. Sind z.B. beide Bits nach der durch CMP ausgeführten Subtraktion (D1-2) gelöscht, so ist das Ergebnis positiv (N-Bit=0) geblieben (V-Bit=0): die Bedingung ist nicht erfüllt.

Die Bedingungen EQ und NE sind auch für andere Anwendungen wichtig. So kann man z.B. feststellen, ob bestimmte Bits eines Datenwortes gesetzt sind, indem man folgende Sequenz schreibt:

```
...
AND #%00001111,D1    ;Bits ausmaskieren
BEQ KEINES           ;verzweigt, wenn keines der 4 unteren
                      Bits gesetzt ist
CMP #%00001111,D1
BEQ ALLE             ;verzweigt, wenn alle 4 Bits gesetzt sind
```

Der AND-Befehl bewirkt, daß alle Bits von D1 mit den Bits des Parameters (hier: #%00001111) verglichen werden. Sind in beiden Bytes die gleichen Bits gesetzt, so werden diese in das Ergebnis übernommen. Ist auch nur eines der beiden Bits gelöscht, so ist dies auch im Ergebnis Null. Dadurch ergibt sich, daß in D1 nach der AND-Verknüpfung nur noch diejenigen Bits gesetzt sind, die vorher in den unteren 4 Bits gesetzt waren.

Diese Technik nennt man ausmaskieren. In obigem Beispiel werden die unteren 4 Bits ausmaskiert, d.h. im Ergebnis werden nur diese 4 Bits in den Zustand erscheinen, in dem sie waren. Alle anderen Bits werden durch die AND-Verknüpfung gelöscht. Man kann hier natürlich beliebige Bitkombinationen verwenden.

Ist kein Bit im Ergebnis mehr gesetzt, so wird dies durch das Zero-Flag signalisiert, wodurch der BEQ-Befehl erfüllt wird und verzweigt. Andernfalls wird mit dem nächsten Befehl fortgeführt, in dem D1 mit %00001111 verglichen wird. Wenn beides gleich ist, so waren ursprünglich mindestens alle der unteren 4 Bits gesetzt und der folgende BEQ-Befehl verzweigt.

Die Bedingungen CC und CS kann man außer beim CMP auch noch verwenden, um beim Rotieren von Daten mittels der ROL- bzw. ROR-Befehle festzustellen, wenn ein HI-Bit aus dem Datenwort herausgeschoben wurde.

Bevor wir nun als nächstes an den Befehlssatz des MC68000 gehen, möchte ich Ihnen noch einen Tip geben:

Mit dem K-SEKA-Assembler ist man auf einfachste Weise in der Lage, jeden Befehl in all seinen Möglichkeiten auszuprobieren. Nehmen wir als Beispiel den eben besprochenen CMP-Befehl. Um diesen Befehl mit verschiedenen Werten zu testen und die Ergebnisse der Vergleiche anhand der Flags direkt zu bekommen, laden Sie bitte den SEKA-Assembler.

Nach der Eingabe der Workspace, z.B. 100, drücken Sie nun die Escape-Taste und gelangen so in den Editor. Nun geben Sie ein (die Zeilennummern gibt der Assembler vor):

```
1 run:
2       cmp  #10,d1
3       bra  run
```

Nun wieder Escape drücken und mit 'a' und zweimal Return dieses Programm assemblieren lassen. Danach mit

```
>xpc <Return>
$000000 run <Return>
```

den Programmzähler auf die Adresse 'run' stellen. Jetzt können Sie mit 'xd1' und dem gewünschten Wert das Datenregister D1 einstellen und dann mit 's' den CMP-Befehl ausführen lassen. Sie können nun direkt den Inhalt der Register (D1 hat sich in diesem Beispiel natürlich nicht verändert) und auch der Flags (neben der Angabe des Statuswortes SR) im Klartext ablesen. Um danach einen anderen Wert für D1 auszuprobieren, können Sie diesen wieder mit 'xd1' einstellen und mit 's2' den BRA-Befehl, welcher kein Register beeinflußt, und erneut den CMP-Befehl laufen lassen.

Übrigens: Dieses Programm sollten Sie nicht mit 'g run' starten. Es läuft dann ewig bzw. bis zum Reset, was nicht gerade erwünscht ist...

Diese Methode ist natürlich nicht auf den CMP-Befehl beschränkt. Ebenso kann damit jeder andere Befehl ausprobiert werden. Sollten Sie dabei einen Fehler machen, so zeigt Ihnen der SEKA dies an, da er alle Exceptions abfängt und sie im Klartext ausgibt (z.B. Adress Error bei MOVE #1,1001). Es ist auch zu empfehlen, dies bei den meisten Befehlen zu tun, da Sie auf diese Weise wesentlich einfacher die Funktion der einzelnen Befehle verstehen können als nur durch das Lesen des nächsten Kapitels.

2.4 Der Befehlssatz

Jetzt ist es wohl langsam an der Zeit, die Befehle des MC68000 zu beschreiben. Für eine erschöpfende Beschreibung jedes einzelnen Befehls ist hier leider nicht der Platz, so daß ich dafür auf die spezielle Literatur über den MC68000 verweise, z.B. das 'Prozessor-Buch zum 68000' von DATA BECKER. In diesem Buch erfahren Sie auch gleich den hardwaremäßigen Aufbau des Prozessors und seiner Peripherie.

Bei den Befehlen sind in den folgenden Tabellen jeweils die benötigten Parameter bzw. Argumente aufgelistet. Die Schreibweise der Argumente bedeutet:

Label	ein Label bzw. eine Adresse
Reg	Register
An	Adreßregister n
Dn	Datenregister n
Quelle	Quelloperand
Ziel	Zieloperand
<ea>	Adresse oder Register
#n	Direktwert

Hier nun eine Liste der Programmsteuerbefehle des MC68000:

Mnemonic		Bedeutung
Bcc	*Label*	Verzweige bedingt, je nach der Bedingung
BRA	*Label*	Verzweige unbedingt (ähnlich wie JMP)
BSR	*Label*	Verzweige in ein Unterprogramm. Ebenso wie bei JSR wird hierbei die Rücksprungadresse auf dem Stack abgelegt und bei RTS zurückgekehrt.
CHK	*<ea>,Dx*	Prüfe ein Datenregister auf Grenzen, löse ggf. die CHK-Instruction-Exception aus
DBcc	*Reg,Label*	Prüfe Bedingung, dekrementiere und verzweige. Dieser Befehl wird häufig für Schleifen verwendet.
JMP	*Label*	Springe zur Adresse (ähnlich wie BRA)
JSR	*Label*	Springe an ein Unterprogramm. Auch hier wird die Rücksprungadresse auf den Stack gelegt, RTS führt zurück.
NOP		Keine Funktion (No Operation)
RESET		Peripherie zurücksetzen (Vorsicht!)
RTE		Rückkehr aus einer Exception
RTR		Rückkehr mit Laden der Flags
RTS		Rückkehr aus einem Unterprogramm (nach BSR oder JSR)
Scc	*<ea>*	Setze ein Byte auf -1, wenn die Bedingung erfüllt ist
STOP		Verarbeitung anhalten (Vorsicht!)
TRAP	*#n*	Springe in eine Exception
TRAPV		Prüfe, ob Überlauf-Flag gesetzt, dann evtl. TRAPV-Exception

Hierzu einige Bemerkungen:

Beim Sprung (JSR) bzw. der Verzweigung (BSR) in eine Unterroutine wird die Rücksprungadresse, wohin nach Beendigung der Unterroutine zurückgekehrt werden soll, auf dem Stack abgelegt. Bei RTS wird einfach diese Adresse vom Stack geholt und dorthin gesprungen.

Wir wollen damit auch einmal ein wenig herumexperimentieren. Dafür schreiben Sie bitte im SEKA folgendes 'Programm':

```
1 run:
2         pea     unterprogramm     ;Adresse auf den Stapel
3         jsr     unterprogramm     ;Unterprogramm-Aufruf
4         move.l  (sp)+,d1          ;Langwort vom Stapel in D1
5         illegal                   ;Abbruch im SEKA
6
7 unterprogramm:
8         move.l  (sp),d0           ;Rücksprungadresse in D0
9         rts                       ;und zurück
```

Schauen wir uns dieses Programm einmal genauer an. Als erstes wird durch den Befehl PEA die Adresse des Unterprogramms auf den Stapel gelegt. Danach wird mit dem JSR-Befehl das Unterprogramm aufgerufen. Die Rücksprungadresse, also diejenige Adresse, an die nach Ablauf des Unterprogramms das Hauptprogramm weiterlaufen soll, wird auf dem Stack abgelegt.

In dem Unterprogramm wird nun das Langwort, auf das der Stackpointer zeigt, in das Datenregister D0 geladen. Danach wird durch den RTS-Befehl die Rücksprungadresse wieder vom Stapel geholt und dorthin gesprungen.

Wieder zurück im Hauptprogramm wird nun das Langwort, welches oben auf dem Stapel liegt, heruntergeholt und in D1 gebracht. Danach bricht der SEKA, wenn das Programm von ihm aus aufgerufen wurde, durch den ILLEGAL-Befehl ab und zeigt die Inhalte der Register.

Assemblieren Sie nun mit 'a' und zweimal Return dieses Programm und starten Sie es mit 'g run' und zweimal Return. Der

SEKA meldet sich sofort wieder mit 'Illegal Instruction' und der Angabe der Registerinhalte. Hier können Sie dann erkennen, daß in D0 die Adresse liegt, an der nach dem RTS-Befehl weitergearbeitet werden soll. Außerdem enthält D1 die Adresse des Unterprogramms, was Sie durch '? unterprogramm' und Vergleichen des Ergebnisses mit dem Inhalt von D1 überprüfen können. Um dies auch direkt vergleichen zu können, geben Sie nun einfach 'n run' ein, woraufhin Sie Ihr Programm disassembliert bekommen.

Eine weitere interessante Information, welche die Statuszeile des SEKA-Assemblers beinhaltet, ist die der Stapelzeigerwerte. Wie Sie wissen, enthält das Adreßregister A7 diesen Zeiger. Doch warum gibt der SEKA noch einmal diesen Wert aus, obwohl man ihn am Adreßregister A7 ablesen kann?

Wie Sie sehen, werden zwei Stapelzeigerwerte ausgegeben: USP und SSP. Da wir uns im Moment noch im User-Modus befinden, sind der Wert von A7 und des USP identisch. Nun wollen wir einmal den anderen Fall ausprobieren.

Dafür geben wir 'x sr' ein und erhalten als Eingabe-Aufforderung, auch Prompt genannt, den Inhalt des Statusregisters ausgegeben. Wir geben nun '$2000' ein. Wenn Sie nun nocheinmal 'x' eingeben, sehen Sie, daß neben der Anzeige des Statusregisters (SR) der Buchstabe 's' steht. Dies bedeutet, daß der Rechner sich nun im Supervisor-Modus befindet.

Starten Sie nun bitte noch einmal das Programm und sehen sich dann den Inhalt von D1, A7 und USP an. A7 und D1 sind wieder so wie vorhin, A7 weicht jedoch vom USP-Wert ab. Der aktuelle Stapelzeiger ist jetzt nämlich nicht mehr der USP, sondern der SSP, da wir uns im Supervisor-Modus befinden. Dies sehen Sie nicht nur an dem kleinen 's' in der Statuszeile, sondern auch daran, daß A7 und SSP denselben Wert anzeigen.

Doch nun zurück zu der Betrachtung der oben aufgeführten Steuerbefehle des MC68000.

STOP und RESET sind so tiefgreifende Befehle, daß sie nur privilegiert anzuwenden sind (aus dem Supervisor-Modus). Auch wenn Sie mit dem SEKA in den Supervisor-Modus umschalten, sollten Sie diese Befehle nicht ausprobieren, zumindest dann nicht, wenn Sie noch wichtige Daten im Rechner haben, die noch nicht abgespeichert sind. (Rechnen Sie mit dem Guru!)

Der TRAP-Befehl bekommt zusätzlich eine Nummer zwischen 0 und $F, welche den zu verwendenden TRAP-Vektor (Adressen $0080-$00BF) und damit die aufzurufende Exception-Routine bestimmt. Einige Betriebssysteme auf 68000-Basis verwenden diesen Befehl zum Aufruf von Betriebssystem-Funktionen. Mit diesem Befehl werden wir uns später noch befassen.

In dem kleinen Beispielprogramm, in dem wir zwei Zahlen verglichen haben, wurde beim CMP eine arithmetische Operation, eben die Subtraktion, durchgeführt, welche natürlich auch mit einem richtigen Ergebnis möglich ist (SUB-Befehl). Dem gegenüber steht natürlich auch die Möglichkeit der Addition (ADD-Befehl). In diesen beiden Operationen erschöpft sich die Rechenfähigkeit eines üblichen 8-Bit-Prozessors wie dem 6502. Der MC68000 kann aber noch mehr. Er kann zusätzlich noch multiplizieren, dividieren und all diese Operationen mit verschiedenen Datengrößen durchführen.

Für die meisten dieser Funktionen werden natürlich zwei Parameter benötigt, so z.B. beim ADD-Befehl. Die Schreibweise lautet daher:

```
ADD Quelle,Ziel
```

wobei Quelle und Ziel Register oder Speicheradressen sein können, Quelle auch ein direkter Wert (#n). Das Ergebnis der Operation liegt dann im Zielregister bzw. an der Zielspeicherstelle. Dies gilt für alle Operationen dieser Art. Auch diese Befehle können Sie anhand des SEKA-Assemblers direkt ausprobieren, wobei als Ziel die Verwendung eines Registers empfehlenswert ist.

Hier nun eine Übersicht über die arithmetischen Operationen mit ganzen Zahlen:

Mnemonic		Bedeutung
ADD	*Quelle,Ziel*	Binäre Addition
ADDA	*Quelle,An*	Binäre Addition zu einem Adreßregister
ADDI	*#n,<ea>*	Addition mit einer Konstanten
ADDQ	*#n,<ea>*	Schnelle Addition einer Konstanten, welche nur von 1 bis 8 sein darf
ADDX	*Quelle,Ziel*	Addition mit Übertrag im X-Flag
CLR	*<ea>*	Löschen eines Operanden
CMP	*Quelle,Ziel*	Vergleich zweier Operanden
CMPA	*<ea>,An*	Vergleich mit einem Adreßregister
CMPI	*#n,<ea>*	Vergleich mit einer Konstanten
CMPM	*Quelle,Ziel*	Vergleich zweier Speicheroperanden
DIVS	*Quelle,Ziel*	Vorzeichenrichtige Division des 32-Bit Ziel- durch den 16-Bit-Quelloperanden. Das Ergebnis der Division liegt danach im LO-Wort des Ziels, der Rest der Division im HI-Wort.
DIVU	*Quelle,Ziel*	Division ohne Vorzeichen, ähnlich wie DIVS
EXT	*Dn*	Vorzeichenrichtige Erweiterung auf doppelte Breite
MULS	*Quelle,Ziel*	Vorzeichenrichtige Multiplikation zweier Worte zu einem Langwort
MULU	*Quelle,Ziel*	Multiplikation ohne Vorzeichen, ähnlich wie MULS
NEG	*<ea>*	Negation eines Operanden (Zweierkomplement)

NEGX	*<ea>*	Negation eines Operanden mit Übertrag
SUB	*Quelle,Ziel*	Binäre Subtraktion
SUBA	*<ea>,An*	Binäre Subtraktion von einem Adreßregister
SUBI	*#n,<ea>*	Subtraktion einer Konstanten
SUBQ	*#n,<ea>*	Schnelle Subtraktion einer 3-Bit-Konstanten
SUBX	*Quelle,Ziel*	Subtraktion mit Übertrag im X-Flag
TST	*<ea>*	Testen eines Operanden und ggf. setzen von N- und Z-Flag

Zusätzlich zu der Verarbeitung von ganzen Zahlen kann der Prozessor auch direkt mit den sogenannten BCD-Zahlen operieren. BCD steht für Binary Coded Decimal und bedeutet, daß hier Dezimalzahlen verarbeitet werden. Bei dieser Methode werden in jedem Halbbyte nur Ziffern zwischen 0 und 9 gespeichert, so daß man diese Zahlen gut verarbeiten kann. Dafür gibt es folgende Befehle:

Mnemonic		**Bedeutung**
ABCD	*Quelle,Ziel*	Addition zweier BCD-Zahlen
NBCD	*Quelle,Ziel*	Negation einer BCD-Zahl (Neunerkomplement)
SBCD	*Quelle,Ziel*	Subtraktion zweier BCD-Zahlen

Auch diese Befehle empfehle ich auszuprobieren, da die Handhabung von BCD-Zahlen zwar relativ einfach, jedoch ungewohnt ist. Achten Sie dabei jedoch darauf, daß Sie bei Eingaben mit 'x' auch wirklich BCD-Zahlen eingeben, da sonst die Ergebnisse der Operationen nicht korrekt sind. Am einfachsten können Sie dies sicherstellen, indem Sie die Zahlen zwar dezimal, jedoch mit vorgestelltem '$'-Zeichen eingeben.

Als nächstes nun die logischen Operationen, die wir teilweise auch vom BASIC her kennen. Mit diesen Operationen können zwei Werte binär, also Bit für Bit, verknüpft werden.

Mnemonic		Bedeutung
AND	*Quelle,Ziel*	Logisch UND
ANDI	*#n,<ea>*	Logisch UND mit einer Konstanten
EOR	*Quelle,Ziel*	Exklusiv-ODER
EORI	*#n,<ea>*	Exklusiv-ODER mit einer Konstanten
NOT	*<ea>*	Inversion eines Operanden
OR	*Quelle,Ziel*	Logisch ODER
ORI	*#n,<ea>*	Logisch ODER mit einer Konstanten
TAS	*<ea>*	Prüfe ein Byte und setze Bit 7

Das Ergebnis, das bei einer solchen Verknüpfung herauskommt, können Sie beim SEKA auch direkt erfahren, indem Sie z.B.

```
?$55 & $AF
```

eingeben und in diesem Fall $05 erhalten.

Auch einzelne Bits können direkt manipuliert werden. Dafür stehen folgende Befehle zur Verfügung:

Mnemonic		Bedeutung
BCHG	*#n,<ea>*	Verändere Bit n (aus 0 wird 1 und umgekehrt)
BCLR	*#n,<ea>*	Lösche Bit n
BSET	*#n,<ea>*	Setze Bit n
BTST	*#n,<ea>*	Teste Bit n, das Ergebnis steht im Z-Flag

Diese Befehle sind vor allem bei der Bedienung und Auswertung von Daten wichtig, die von einem Peripheriebaustein kommen. Bei diesen Daten spielen nämlich oft die einzelnen Bits jeweils eine besondere Rolle. Sie werden dies noch in späteren Kapiteln genauer kennenlernen.

Des weiteren kann der Prozessor einen Operanden noch in sich verschieben und rotieren (n bedeutet ein Register oder Direktwert mit #, der die Anzahl der Verschiebungen angibt):

Mnemonic		**Bedeutung**
AS	*n,<ea>*	Arithmetische Verschiebung nach links (*2^n)
ASR	*n,<ea>*	Arithmetische Verschiebung nach rechts (/2^n)
LSL	*n,<ea>*	Logische Verschiebung links
LSR	*n,<ea>*	Logische Verschiebung rechts
ROL	*n,<ea>*	Rotation nach links
ROR	*n,<ea>*	Rotation nach rechts
ROXL	*n,<ea>*	Rotation nach links mit Übertrag ins X-Flag
ROXR	*n,<ea>*	Rotation nach rechts mit Übertrag ins X-Flag

All diese Befehle dienen dazu, ein Byte, Wort oder Langwort binär nach links bzw. rechts zu schieben. Warum dies gleichzeitig eine Multiplikation bzw. Division durch eine Zweierpotenz 2^n bewirkt, kann man leicht an Hand der binären Darstellung eines Beispiels erkennen.

Wir nehmen als Beispiel ein Byte mit dem Inhalt 16. Binär dargestellt ist dies:

```
%00010000 = 16
```

Nun schieben wir dieses Byte einmal nach links, wobei wir rechts eine 0 einschieben. Das Ergebnis:

```
%00010000 verschoben nach links ergibt
%00100000 = 32, also 16*2
```

Mehrmaliges Schieben ergibt somit auch mehrmaliges verdoppeln, wodurch bei n-mal Schieben effektiv mit 2^n multipliziert wird.

Dasselbe gilt natürlich auch für die Verschiebung nach rechts. Hierbei ist allerdings eine Besonderheit zu beachten. Als Beispiel ein Byte mit dem Inhalt 5:

```
%00000101 = 5, einmal verschoben nach rechts ergibt
%00000010 = 2
```

Wie Sie sehen, kommt hier nicht 2,5 heraus. Wie soll dies auch dargestellt werden? Das Ergebnis einer solchen Division ergibt also immer nur einen ganzzahligen Wert, eventuelle Nachkommastellen entfallen. Verwendet man dagegen den DIV-Befehl, so erhält man neben dem Ergebnis auch den Rest der Division, so daß dabei nichts verlorengeht. Die Vorteile der Verschiebe-Methode gegenüber dem DIV-Befehl sind jedoch, daß sie erstens schneller ist und zweitens auch ganze Langworte als Ergebnis erhalten kann.

Nach der Klärung des Prinzips von Verschiebungen bleibt nun die Frage, wieso hierfür mehr als 2 Befehle nötig sind. Dies liegt daran, daß es mehrere Arten von Verschiebungen gibt.

Zunächst einmal unterscheidet man zwischen Verschieben, auf Englisch 'to shift', und Rotieren. Der Unterschied ist lediglich der, daß beim Verschieben immer eine 0 in das freiwerdende Bit eingeschoben wird. Beim Rotieren dagegen wird ein bestimmter Wert eingeschoben. So wird beim Befehl ROR bzw. ROL das unten bzw. oben herausgeschobene Bit auf der anderen Seite wieder hineingeschoben, beim ROXR- bzw. ROXL-Befehl nimmt dieses Bit den Umweg über das X-Flag. Der Inhalt dieses Flags wird also in das rotierte Wert hineingeschoben und das herausgeschobene Bit in das Flag geladen.

Auch beim Schieben gibt es zwei Variationen: arithmetische und logische Verschiebung. Die logische Verschiebung haben wir bereits kennengelernt: es wird immer eine Null eingeschoben und das letzte herausgeschobene Bit steht sowohl im C- als auch im X-Flag.

Bei der arithmetischen Verschiebung dagegen wird das höchstwertige Bit, welches ja das Vorzeichen darstellt, zwar mitgeschoben, jedoch bei ASR beibehalten. Dies hat den Vorteil, daß bei Verwendung dieser Befehle zur Division das Vorzeichen bleibt, so daß auch z.B. -10/2 auch -5 ergibt. Ein dennoch auftretender Vorzeichenwechsel durch Über- bzw. Unterlauf wird durch das entsprechende Flag, das V-Flag, gemeldet, welches bei einem Vorzeichenwechsel gesetzt wird. Bei der logischen Verschiebung wird dieses Flag immer gelöscht.

Und nun kommen wir zu den Befehlen, die Daten im Rechner bewegen. Dies sind eigentlich sogar die wichtigsten Befehle eines jeden Prozessors, denn wie sollen sonst Daten verarbeitet werden können?

Mnemonic		**Bedeutung**
EXG	*Rn,Rn*	Austauschen zweier Registerinhalte (nicht mit SWAP verwechseln!)
LEA	*<ea>,An*	Lade eine effektive Adresse in Adreßregister An
LINK	*An,#n*	Baue Stackbereich auf
MOVE	*Quelle,Ziel*	Übertrage einen Wert von Quelle nach Ziel
MOVE	*SR,<ea>*	Übertrage den Statusregister-Inhalt
MOVE	*<ea>,SR*	Übertrage den Statusregister-Inhalt
MOVE	*<ea>,CCR*	Flags laden
MOVE	*USP,<ea>*	Übertrage den User-Stackpointer
MOVE	*<ea>,USP*	Übertrage den User-Stackpointer

MOVEA	*<ea>,An*	Übertrage einen Wert in das Adreßregister An
MOVEM	*Regs,<ea>*	Übertrage mehrere Register auf einmal
MOVEM	*<ea>,Regs*	Übertrage mehrere Register auf einmal
MOVEP	*Quelle,Ziel*	Übertrage Daten zur Peripherie
MOVEQ	*#n,Dn*	Übertrage schnell eine 8-Bit-Konstante in das Datenregister Dn
PEA	*<ea>*	Lege eine Adresse auf dem Stack ab
SWAP	*Dn*	Vertausche die Registerhälften (die oberen gegen die unteren 16 Bit)
UNLK	*An*	Baue Stackbereich ab

Die Befehle LEA oder PEA sind sehr beliebt, um Adressen in ein Adreßregister zu laden bzw. auf den Stapel zu legen. Die Anweisung

```
LEA  Label,A0
```

lädt die Adresse des Labels 'Label' in das Adreßregister A0. Dies entspricht in der Funktion genau dem Befehl

```
MOVE.L  #Label,A0
```

Dementsprechend bedeutet

```
PEA  Label
```

nichts anderes, als daß die Adresse des Labels auf den Stapel gelegt wird. Dies erledigt ebenso der Befehl

```
MOVE.L  #Label,-(SP)
```

Interessant wird der LEA-Befehl, wenn man als effektive Adresse nicht einfach ein Label angibt, sondern z.B. eine durch indirekte Adressierung erzeugte Adresse. Ein Beispiel:

```
LEA  1(A0,D0),A1
```

Die Adresse, die bei obiger Adressierung durch die Addition von 1 (Direktwert-Offset)+A0+D0 erzeugt wird, liegt dann in A1. Dieses Beispiel läßt sich nicht mit nur einem MOVE-Befehl erzeugen. Ein entsprechendes Programmteil wäre:

```
MOVE.L  A0,A1
ADD.L   D0,A1
ADDQ.L  #1,A1
```

Sie sehen, der LEA-Befehl bietet interessante Möglichkeiten!

Das waren alle Befehle des MC68000. Durch die Kombination mit den diversen Adressierungsarten lassen sich sehr viele Befehle zusammenstellen, um ein Programm so effizient wie möglich zu machen.

Die nun folgende Tabelle zeigt noch einmal alle Befehle des MC68000-Prozessors mit den möglichen Adressierungsarten und der Beeinflussung der Flags. Die verwendeten Symbole bedeuten:

x=erlaubt	q=nur für Quelle	z=nur für Ziel
-=unbeeinflußt	0=gelöscht	*=entspr. modifiziert
1=gesetzt	u=unbestimmt	P=privilegiert

Mnemonic	1	2	3	4	5	6	7	8	9	10	11	12	XNZVC	P
ABCD	x				x									
ADD	q	q	x	x	x	x	x	x	x	q	q	q	*****	
ADDA	x	x	x	x	x	x	x	x	x	x	x	x	-----	
ADDI	x		x	x	x	x	x	x	x				*****	
ADDQ	x	x	x	x	x	x	x	x	x				*****	
ADDX	x				x								*****	
AND	q		x	x	x	x	x	x	x	q	q	q	-**00	
ANDI	x		x	x	x	x	x	x	x				-**00	
ASL, ASR	x		x	x	x	x	x	x	x				*****	

Mnemonic	1	2	3	4	5	6	7	8	9	10	11	12	XNZVC	P
Bcc													-----	
BCHG	x		x	x	x	x	x	x	x				--*--	
BCLR	x		x	x	x	x	x	x	x				--*--	
BRA													-----	
BSET	x		x	x	x	x	x	x	x				--*--	
BSR													-----	
BTST	x		x	x	x	x	x	x	x	z	x	x	--*--	
CHK	x		x	x	x	x	x	x	x	x	x	x	-*uuu	
CLR	x		x	x	x	x	x	x	x				-0100	
CMP	x	x	x	x	x	x	x	x	x	x	x	x	-****	
CMPA	x	x	x	x	x	x	x	x	x	x	x	x	-****	
CMPI	x		x	x	x	x	x	x	x				-****	
CMPM			x			x	x	x	x		x	x	-***	
cpGEN													-----	
DBcc													-----	
DIVS	x		x	x	x	x	x	x	x	x	x	x	-***0	
DIVU	x		x	x	x	x	x	x	x	x	x	x	-***0	
EOR	x		x	x	x	x	x	x	x				-**00	
EORI	x		x	x	x	x	x	x	x				-**00	
EORI CCR													*****	
EORI SR													*****	
EXG													-----	
EXT													-**00	
EXTB													-**00	
ILLEGAL													-----	
JMP			x			x	x	x	x		x	x	-----	
JSR			x			x	x	x	x		x	x	-----	
LEA			x			x	x	x	x		x	x	-----	
LINK		x											-----	
LSL, LSR			x	x	x	x	x	x	x				***0*	
MOVE	x	q	x	x	x	x	x	x	x	q	q	q	-**00	
MOVEA	x	x	x	x	x	x	x	x	x	x	x	x	-----	
MOVE to CCR	x		x	x	x	x	x	x	x	x	x	x	*****	
MOVE from SR	x		x	x	x	x	x	x	x				-----	P
MOVE to SR	x		x	x	x	x	x	x	x	x	x	x	*****	P
MOVE USP		x											-----	P

Mnemonic	1	2	3	4	5	6	7	8	9	10	11	12	XNZVC	P
MOVEM			x	q	z	x	x	x	x		q	q	-----	
MOVEP	q	z											-----	
MOVEQ	z												-**00	
MULS	x		x	x	x	x	x	x	x	x	x	x	-**00	
MULU	x		x	x	x	x	x	x	x	x	x	x	-**00	
NBCD	x		x	x	x	x	x	x	x				*u*u*	
NEG	x		x	x	x	x	x	x	x				*****	
NEGX	x		x	x	x	x	x	x	x				*****	
NOP													-----	
NOT	x		x	x	x	x	x	x	x				-**00	
OR	q		x	x	x	x	x	x	x	q	q	q	-**00	
ORI	x		x	x	x	x	x	x	x				-**00	
PEA			x			x	x	x	x		x	x	-----	
RESET													-----	P
ROL, ROR			x	x	x	x	x	x	x				-**0*	
ROXL, ROXR			x	x	x	x	x	x	x				-**0*	
RTE													-----	P
RTR													*****	
RTS													-----	
SBCD	x				x								*u*u*	
Scc	x		x	x	x	x	x	x	x				-----	
STOP										x			-----	
SUB	q	q	x	x	x	x	x	x	x	q	q	q	*****	
SUBA	x	x	x	x	x	x	x	x	x	x	x	x	-----	
SUBI	x		x	x	x	x	x	x	x				*****	
SUBQ	x	x	x	x	x	x	x	x	x				*****	
SUBX	x				x								*****	
SWAP	x												-**00	
TAS	x		x	x	x	x	x	x	x				-**00	
TRAP										x			-----	
TRAPV													-----	
TST	x		x	x	x	x	x	x	x				-**00	
UNLK		x											-----	

3. Die Arbeit mit Assemblern

All diese Befehle für den MC68000-Prozessor, welche wir kennengelernt haben, sind für ihn in dieser Form nicht verständlich. So sagen dem Prozessor die Buchstaben 'MOVE' überhaupt nichts; er braucht seine Befehle in binärer Form. Jeder Befehl muß also in einem Wort codiert werden, was normalerweise viel Arbeit bedeutet.

Diese Arbeit übernimmt ein sogenannter Assembler. Ein solches Programm übersetzt die Befehle, welche in Textform vorliegen, in die entsprechenden binären Befehlsworte. Den Text, welchen wir dort eingaben, bezeichnet man als Mnemonic oder Mnemocode. Dieser Text ist wirklich bedeutend einfacher zu lesen, oder sagt Ihnen $4280 mehr als CLR.L D0 ?

In diesem Kapitel soll nun die Arbeit mit den verschiedenen Assemblern beschrieben werden. Dabei werden beschrieben:

ASSEM

Der Assembler des Amiga-Entwicklungspaketes. Dieser Assembler ist recht leistungsfähig, wird aber in einigen Dingen klar von den beiden anderen Kollegen übertroffen.

Profimat Amiga

Der DATA BECKER-Assembler, welcher bereits in der ATARI-ST-Version sehr beeindruckt hat. Er beinhaltet außer dem Assembler noch einen sogenannten Debugger, mit dem bestehende Programme getestet und ggf. korrigiert werden können.

KUMA-SEKA

Der bewährte Assembler, ebenfalls mit einem Debugger ausgestattet, der sich meiner Meinung nach am besten eignet, Übungsprogramme zu schreiben und damit zu experimentieren. Aus diesem Grund sind die Programme in diesem Buch mit diesem Assembler geschrieben.

Allen Assemblern ist gemeinsam, daß sie ein im Mnemocode geschriebenes Programm übersetzen und als lauffähiges Pro-

gramm auf die Diskette schreiben können. Um die übersetzten Programme direkt ausprobieren zu können, benötigt man allerdings einen Debugger, den der erste der hier besprochenen Assembler nicht besitzt.

3.1 Der Assembler des Entwicklungspaketes

Dieser Assembler gehört zu den reinen Disketten-Assemblern. Dies bedeutet, daß er nur assemblieren kann, was als Textdatei auf Diskette vorliegt und das oder die Ergebnisse wieder auf Diskette schreibt. Es sind hier also weder direkte Eingaben noch Probeläufe des neuen Programms möglich.

Der ASSEM wird vom CLI aus aufgerufen, indem man seinen Namen und dahinter die benötigten Parameter eingibt.

Der Aufruf erfolgt im einfachsten Fall so:

```
ASSEM Quelle -O Ziel
```

wobei Quelle der Dateiname der Quelldatei mit dem Programmtext und Ziel der Dateiname derjenigen Datei ist, in welchem das Ergebnis der Assemblierung abgespeichert werden soll. Das -O bedeutet, daß der folgende Dateiname für die zu erzeugende Objektdatei verwendet werden soll.

Es gibt noch einige weitere Möglichkeiten, Parameter mit anzugeben. Diese werden dann jeweils mit ihrer Option zusammen geschrieben (wie -O), damit der Assembler weiß, was er mit der dort angegebenen Datei anzufangen hat. Diese möglichen Optionen, auf die jeweils der Dateiname folgen muß, sind:

-O Objektdatei (s.o.)

-V in diese Datei werden alle Meldungen, die bei der Assemblierung erfolgen (z.B. Fehlermeldungen) geschrieben. Ist hier keine angegeben, so erfolgen diese Meldungen im CLI-Fenster.

-L Die Ausgabe der assemblierten Programmzeilen wird in diese Datei geleitet. Es kann hier auch z.B. PRT: angegeben werden, damit gedruckt wird.

-H Diese Datei wird an den Anfang der zu assemblierenden Datei gelesen und mit assembliert.

-E Diese Datei wird erzeugt und erhält dann die Zeilen, in denen EQU-Direktiven stehen.

-C Diese Option wird ohne Dateinamen, jedoch mit einer weiteren Option angegeben. Sie kann auch als OPT geschrieben werden. Ihr folgt dann das Zeichen:

OPT S Es wird eine Symboltabelle erzeugt, in der alle Labels mit ihrem Wert aufgeführt werden.

OPT X Es wird eine Liste mit Querverweisen erzeugt (wo wird welches Label verwendet?).

OPT W Hiernach muß eine Zahl folgen, welche den zu reservierenden Arbeitsspeicherbereich festlegt.

Der Assembler erzeugt dann schließlich eine sogenannte Objektdatei, welche jedoch noch nicht lauffähig ist. Soll sie lauffähig gemacht werden, muß man noch den Linker namens ALINK aufrufen. Dieses Programm ist in der Lage, mehrere assemblierte oder auch compilierte Objektdateien zusammenzubinden und zu einem lauffähigen Programm zu machen. Im einfachsten Fall schreiben Sie folgendes Kommando in den CLI:

```
ALINK Quelle TO Ziel
```

Quelle ist dabei das durch den Assembler erzeugte Objektdatei, Ziel ist der Programmname, welcher schließlich direkt gestartet werden kann.

3.2 Der Profimat Amiga

Der Profimat Amiga aus dem Hause DATA Becker stellt eine gelungene Kombination von Editor, Assembler und Debugger dar, welche einfach zu bedienen ist. Leider lag zur Zeit der Erstellung dieses Buches keine endgültige Version dieses Programms vor, jedoch wurde mir vom Autor des Programms versichert, daß die Amiga-Version auf ähnliche Art und Weise funktionieren wird wie die ATARI ST-Version, mit der ich bereits gute Erfahrung gemacht habe.

Der Profimat ist in mehrere Fenster unterteilt, je eines für den Assembler, den Editor, den Debugger und einige Hilfsfunktionen. Die Vorgehensweise bei der Erstellung eines Programms ist somit recht einfach:

1. Mit dem Editor das Programm schreiben und dann erst einmal auf Diskette abspeichern.
2. Den Assembler starten und so das Programm übersetzen lassen.
3. Bei Bedarf den Debugger starten, das erstellte Programm einladen und testen.

In dem Debugger haben Sie dann auch die Möglichkeit, nur Programmteile oder auch nur einzelne Befehle abarbeiten zu lassen. Da auch nach jedem solchen Schritt die Zustände der Register und der Flags im Statusregister angezeigt werden, können Sie mit diesem Debugger recht leicht die in diesem Buch vorgestellten Programme ausprobieren.

Im Gegensatz zum ASSEM brauchen Sie also für die gesamte Arbeit mit den Maschinenprogrammen nur einmal den Profimat zu laden, ohne immer zwischen Editor, Assembler, Linker und Debugger hin- und herladen zu müssen.

Eine besonders interessante Funktion des Profimat sollte auch erwähnt werden: den Reassembler. Mit dieser Debugger-Funktion können Sie fertige Programme, die Ihnen nur als lauffähiges

Programm vorliegen, in den Quelltext des Programms zurückverwandeln, welchen Sie dann mit dem Editor weiterverarbeiten oder ändern und danach wieder neu assemblieren können.

3.3 Der K-SEKA-Assembler

Und nun last not least den SEKA von der Firma KUMA, welcher außer dem Assembler noch einen einfachen Texteditor und einen Debugger beinhaltet. Dieses Programm ist durch einfache Kommandos zu steuern und ist dadurch sehr handlich. Außerdem ist es so vielseitig und schnell, daß es sich zum Ausprobieren von kleinen Beispiel- und Testprogrammen sehr gut eignet. Größere Programme kann man damit natürlich ebensogut schreiben, wenn man sich mit dem etwas simplen Editor angefreundet hat. Betrachten wir nun als erstes diesen Editor.

Wollen Sie ein Programm als Quellcode (Text) in den Editor laden, so geben Sie einfach 'r' (Read) ein. Es erfolgt dann die Frage nach dem zu ladenden Dateinamen 'FILENAME>', wo Sie nun den Namen der Textdatei eingeben. Wenn Sie diese Datei ebenfalls mit dem SEKA erstellt haben, trägt diese Datei auf der Diskette den gewählten Namen mit dem Anhang '.S'. Zum Laden dieser Datei brauchen Sie dieses .S nicht hinzuzufügen, da dies automatisch geschieht. Das S steht für Source, was auf deutsch Quelle bedeutet.

Das Abspeichern eines erstellten bzw. modifizierten Programms erfolgt durch das Kommando 'w', was wiederum nach dem Namen fragt. Wenn Sie hierauf 'Test' eingeben, wird auf der Diskette die Datei 'Test.S' angelegt. Dies ist einen normale Text-Datei im ASCII-Format.

Um nun ein Programm einzugeben oder zu ändern, bietet der SEKA zwei Möglichkeiten: einen einfachen Zeilen- und einen Bildschirm-Editor. In letzteren gelangen Sie, wenn Sie die Escape-Taste drücken. Der obere Bildschirm- bzw. Fensterteil wird dann für den Editor reserviert, in dem Sie nach Belieben mit dem Cursor herumfahren und Änderungen im Text vornehmen können. Dabei werden eingegebene Zeichen grundsätzlich

in den bestehenden Text eingeschoben und neue Zeilen automatisch mit Nummern versehen. Durch nochmaliges Betätigen der Escape-Taste verlassen Sie den Bildschirm-Editor wieder.

Über diesen Editor braucht eigentlich nicht viel mehr gesagt werden. Er eignet sich wirklich nur für einfache Eingaben und Änderungen. Die weiteren Funktionen werden wieder im normalen Kommandomodus eingegeben, bei dem das '>'-Zeichen als Eingabeaufforderung erscheint.

Folgende Kommandos stehen Ihnen hier für die Textbearbeitung zur Verfügung (<n> steht für eine Zahl, die Bedeutung des Kommandos finden Sie in der runden Klammer):

Kommando (Bedeutung)	**Funktion**
t (Target)	Setzt den Cursor in die oberste Zeile des Textes.
t <n>	Setzt den Cursor in die durch <n> angegebene Zeile
b (Bottom)	Setzt den Cursor in die letzte Zeile des Textes
u (Up)	Eine Zeile hoch
u <n>	<n> Zeilen hoch
d (Down)	Eine Zeile runter
d <n>	<n> Zeilen runter
z (Zap)	Löscht die aktuelle Zeile
z <n>	Löscht <n> Zeilen ab der Cursorzeile
e (Edit)	Erlaubt das Editieren der aktuellen Zeile (und nur die)
e <n>	Editieren von Zeile <n>

f text (Find) (Look	Sucht ab der aktuellen Zeile nach dem angegebenen Text. Groß- und Kleinschreibung wird dabei berücksichtigt und ist daher korrekt einzugeben. Nach dem 'f' eingegebene Leerzeichen werden mitgesucht!
f	Sucht weiter nach dem vorher angegebenen Text
i (Insert)	Startet den Zeilen-Editor. Nun können Sie ein Programm Zeile für Zeile eingeben, allerdings ohne die Möglichkeit, mit dem Cursor in andere Zeilen zu springen. Zeilennummern werden automatisch generiert, folgende Zeilen werden nicht gelöscht, sondern weitergeschoben.
ks (Kill Source)	Nach einer Sicherheitsabfrage (Sure?) wird, wenn sie mit 'y' beantwortet wurde, der Quelltext gelöscht. Andernfalls passiert nichts.
o (Old)	Macht die 'ks'-Funktion wieder rückgängig und rettet somit den alten Text
p (Print)	Gibt die aktuelle Zeile aus
p <n>	Gibt <n> Zeilen ab der Cursorzeile aus

Dies waren die Editorfunktionen des K-SEKA. In Verbindung mit dem Bildschirm-Editor stellen sie eine recht einfache Art der Textbearbeitung dar. So kann man, wenn man im Bildschirm-Editor arbeitet, die aktuelle Zeile und ggf. auch weitere Zeilen löschen, indem man mit Escape in den Kommandomodus springt und direkt 'z' bzw. 'z <n>' eingibt.

Andersherum kann man folgendermaßen vorgehen, wenn man alle Zeilen, die z.B. einen 'trap'-Befehl enthalten, editieren will:

- mit 't' an den Textanfang springen,
- mit 'ftrap' die erste Zeile mit einem 'trap'-Befehl suchen,
- Escape drücken und die Zeile editieren,
- mit erneutem Escape wieder in den Kommandomodus springen,
- mit 'f' weitersuchen, Escape usw., bis das Textende erreicht wurde.

Was hier noch recht umständlich klingt, erweist sich in der Praxis als durchaus brauchbar und schnell. Am besten probieren Sie mit den Editiermöglichkeiten ein wenig herum, damit Sie den Editor sicher bedienen können.

Nun folgen einige Kommandos, die für die Arbeit mit Disketten vorgesehen sind:

Kommando (Bedeutung)	**Funktion**
v (View Files)	Ausgabe des Inhaltsverzeichnisses der Diskette. Es kann hier auch die Angabe eines anderen Laufwerks oder eines Unterdirectories mit übergeben werden, z.B. 'vc' listet den Inhalt des 'c'- Unterdirectories auf und macht dies gleichzeitig zum aktuellen Directory.
kf (Kill File)	Es wird nach einem Filenamen gefragt. Diese Datei wird dann (ohne Sicherheitsabfrage) gelöscht.
r (Read)	Nach der Eingabe dieses Kommandos wird nach dem zu ladenden Dateinamen (FILENAME>) gefragt und diese Datei dann geladen. Wird nur 'r' eingegeben, so wird eine Textdatei in den Editor geladen.

ri (Read Image)	Lädt eine Datei in den Speicher. Nach der Eingabe des Dateinamens fragt SEKA nach der Adresse (BEGIN>), ab der geladen werden soll, und dann nach der Endadresse (END>), bis zu der maximal zu laden ist.
rx (Read from Auxillary)	Entspricht der ri-Funktion, nur daß dabei nicht von der Diskette, sondern vom seriellen Port gelesen wird (Dateiname entfällt somit).
rl (Read Linkfile)	Erlaubt das Einladen einer mit dem SEKA erstellten Linkdatei (Erklärung s.u.). Es erfolgt zuerst eine Sicherheitsabfrage, da durch das Laden einer Linkdatei der Textspeicher gelöscht wird.
w (Write)	Es folgt die Frage nach dem Dateinamen, unter dem der aktuelle Text abgespeichert werden soll. Es wird an diesen Namen automatisch der Anhang .S angefügt, an dem die SEKA-Dateien erkennbar sind.
wi (Write Image)	Speichert einen Speicherbereich auf der Diskette ab, nachdem der Name, Anfang und Ende eingegeben wurden.
wx (Write to Auxillary)	Wie 'wi', nur wird hier über die serielle Schnittstelle ausgegeben.
wl (Write Linkfile)	Speichert nach Abfrage des Namens ein mit der 'l'-Option assembliertes Linkfile auf der Diskette ab. Ist dies nicht vorhanden, erfolgt die Meldung '** Link option not specified'.

Wenn Sie also ein Programm eingetippt bzw. geladen haben, können Sie nun den Assembler aufrufen, um dieses Programm zu übersetzen. Dafür brauchen Sie nur 'a' einzugeben, worauf die Frage nach der Option folgt. Geben Sie hier nur Return ein, so wird normal assembliert, d.h. ohne Ausgabe des Ergebnisses wird das Programm aus dem Speicher in den Speicher übersetzt. Das Ergebnis ist danach direkt ausführbar.

Sie können aber auch eine oder mehrere von folgenden Optionen eingeben:

v bewirkt die Ausgabe des Ergebnisses auf dem Bildschirm

p oder

e leitet die Ausgabe auf den Drucker mit Titelzeile

h läßt die Ausgabe nach jeder Seite stoppen und wartet dann auf einen Tastandruck. Dies ist recht nützlich, um die Ausgabe auf dem Bildschirm zu kontrollieren oder ein neues Blatt in den Drucker einzulegen.

o läßt den Assembler alle möglichen Branch-Befehle optimieren. Dabei wird, wenn möglich, ein .S an den Branch-Befehl gehängt, was den Programmcode kürzer werden läßt. Es treten dabei einige Meldungen auf, welche Sie jedoch ignorieren können.

l bewirkt die Erzeugung von linkfähigem Code, welcher mit 'wl' abgespeichert bzw. mit 'rl' geladen werden kann.

Im Anhang zu dem Listing selbst wird eine Symboltabelle ausgegeben (wenn gewünscht). In dieser Tabelle werden alle Labels mit ihrem Wert sowie Macro-Namen ausgegeben. Ein solches Macro erlaubt die Zusammenfassung mehrerer Befehle zu einem einzigen. Ein Beispiel:

Sie haben eine Routine namens 'pline' geschrieben, welche einen Text ausgibt, auf den das Register A0 zeigt. Sie müssen somit jedesmal schreiben:

```
lea text,a0 ;Zeiger auf Text in A0
bsr pline ;Text ausgeben
```

Dies können Sie sich nun vereinfachen, indem Sie für diese Funktion ein Macro definieren. Dafür schreiben Sie am Anfang des Programms:

```
print: macro ;Macro namens 'print'
 lea ?1,a0 ;Parameter in A0
 bsr pmsg ;Text ausgeben
endm ;Macro-Ende
```

In Ihrem Programm können Sie nun einfach schreiben:

```
print   text        ;Text ausgeben
```

Diese Zeile wird in Wirklichkeit vom Assembler durch das Macro ersetzt. Der Parameter 'text' wird dann dort eingefügt, wo '?1' steht. Sie können auch mehrere Parameter an ein Macro übergeben, welche Sie dann mit '?2' ect. verwerten können.

Sie können nun auch wählen, ob Sie in dem Ausgabelisting des Assemblers die Macros sehen wollen. Dies und vieles mehr wird durch Anweisungen an den Assembler in den Programmtext eingetragen, die Pseudo-Operators (kurz Pseudoops) genannt werden. Der SEKA kennt folgende Pseudoops:

dc Definiert ein oder mehrere Daten, die an dieser Stelle im Programm stehen sollen. Hierbei wird durch .B, .W oder .L die Wortbreite angegeben, wird diese weggelassen, wird .B angenommen. Texte können ebenfalls, in Hochkommas oder Anführungsstrichen, eingegeben werden.

Beispiel: `dc.b "Hallo",10,13,0`

blk Reserviert eine Anzahl Bytes, Worte oder Langworte, je nach Angabe (.B, .W, .L). Der erste Parameter gibt die Anzahl der zu reservierenden Worte an, der zweite (optional) wird zum Füllen des Speicherbereichs verwendet.

Beispiel: `blk.w 10,0`

org Der folgende Parameter gibt die Speicheradresse an, in die das (absolute) Programm assembliert werden soll.

Beispiel: `org $40000`

code Schaltet wieder in den relativen Modus, in dem ein Programm so assembliert wird, daß es an der Adresse 0 beginnt. Die Neuadressierung, je nachdem, wohin das Programm geladen wurde, nimmt dann der Amiga selbst vor.

data Ab hier kommen nur noch Daten, kein Programm mehr (kann entfallen).

even Macht die aktuelle Adresse gerade durch eventuelles Einfügen eines Füllbytes.

odd Gegenstück von 'even', macht die Adresse ungerade

end Beendet hier die Assemblierung.

equ oder = Dient zur Zuweisung von Werten an Labels.

Beispiel: `Wert = 123` bzw. `Wert: equ 123`

list Schaltet die Ausgabe wieder an (nach 'nlist'). Es können folgende Parameter mit angegeben werden und bewirken die Ausgabe von:

c Macro-Aufrufe
d Macro-Definitionen
e Macro-Erweiterungen des Programms
x Code-Erweiterungen

Beispiel: `list e`

nlist Schaltet die Ausgabe ab. Hier können die gleichen Parameter mit angegeben werden wie bei 'list'.

page Bewirkt einen Seitenvorschub auf dem Drucker bzw. den Beginn einer neuen Ausgabeseite.

if Der folgende Parameter entscheidet, ob weiter assembliert werden soll: ist er Null, wird nicht weiter assembliert.

else War der 'if'-Parameter Null, so wird hier weiterassembliert.

endif Ende der bedingten Assemblierung

macro Beginn einer Macro-Definition

endm Ende der Macro-Definition

?n Läßt in einem Macro hier den n-ten Parameter aus der aufrufenden Zeile einfügen.

?0 Generiert für jeden Macro-Aufruf eine neue dreistellige Zahl, welche für lokale Labels sehr nützlich ist.

Beispiel: `x?0: bsr pmsg`

illegal Erzeugt einen illegalen Maschinenbefehl

globl Definiert die folgenden Labels als global, wenn die 'l'-Option des Assemblers gewählt ist.

Wenn Sie Ihr Programm schließlich assembliert haben, liegt der erzeugte Programmcode im Speicher. Mit dem Kommando 'h' können Sie nun erfahren, wie groß und wo im Speicher das Programm zu finden ist. Dabei werden Anfangs- und Endadresse sedezimal und die Länge dezimal ausgegeben (je nach zuletzt ausgeführten Operationen):

Work am Anfang definierter Arbeitsspeicher

Src Text im Speicher

RelC Relokationstabelle des Programms

RelD Relokationstabelle des Datenbereichs

Code erzeugter Programmcode

Data Datenbereich des Programms

Sie finden Ihr Programm also an der Adresse im Speicher, die bei Code angegeben ist. Wenn Sie das Programm starten wollen, ist es lästig, immer diese Adresse einzugeben. Es empfiehlt sich daher, den Anfang des Programms mit einem Label zu kennzeichnen (z.B. 'run:'). Sie können dann mit dem 'g'-Befehl das Programm folgendermaßen starten:

```
g run
```

Der 'g'-Befehl (Go) gehört zu den Debugger-Befehlen des SEKA. Hier eine Übersicht:

x Ausgabe aller Register

xr Ausgabe und Änderung von Register r (z.B. xd0)

gn Sprung zur Adresse n. Es wird daraufhin nach Breakpoints gefragt, Adressen, an denen das Programm abbrechen soll.

jn Wie oben, jedoch wird mit 'JSR' in das Programm gesprungen, welches dann mit 'RTS' enden muß.

qn Ausgabe des Speicherinhaltes ab der Adresse n. Es kann zusätzlich noch die Wortbreite mit angegeben werden.
`Beispiel: q.w $10000`

nn Disassemblierte Ausgabe ab Adresse n

an Direktes Assemblieren ab Adresse n. Es werden dann direkt Programmbefehle eingegeben.

mn Modifizieren des Speicherinhaltes ab Adresse n. Auch hier kann die Wortbreite mit angegeben werden. Abbruch der Eingaben mit Escape.

sn Ausführung des Programmbefehls, auf den der PC zeigt. Bei Angabe von n Ausführung von n Programmschritten.

f Füllen eines Speicherbereichs (wahlweise Angabe der Wortbreite). Alle nötigen Parameter werden einzeln erfragt.

c Kopieren eines Speicherbereichs in einen anderen. Alle nötigen Parameter werden einzeln erfragt.

? Ausgabe des Wertes eines Ausdrucks oder Labels.
`Beispiel: ? run+$1000-256`

a Setzen einer Kommandosequenz, welche das Programm beim Start übermittelt bekommt, als wäre es vom CLI aus mit dieser Sequenz gestartet worden.

! Ausgang aus dem SEKA-Assembler nach Sicherheitsabfrage

Die im Beispiel zu '?' angeführten Rechenoperationen zeigen, daß der SEKA auch rechnen kann. Dies kann man natürlich auch in Programmen anwenden. Es funktionieren folgende Operationen im SEKA:

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
&	logisch UND
!	logisch ODER
~	Exclusiv Oder (EXOR)

Diese Operationen können auch kombiniert werden. Eine weitere Unterscheidung findet bei der Auswahl des verwendeten Zahlensystems statt. Es steht $ für sedezimal, @ für oktal und % für binär, ohne Angabe eines dieser Symbole wird eine Dezimalzahl angenommen.

Doch zurück zum Debugger. Wie erwähnt, wird nach der Eingabe von 'g Adresse' nach sogenannten Breakpoints gefragt. Es können dann bis zu 16 Adressen eingegeben werden, an denen die Programmausführung dann stoppen wird. Gibt man keinen Breakpoint an, sondern drückt direkt Return, so muß das Programm mit einem ILLEGAL-Befehl enden. Endet es dagegen mit einem RTS, so wird an dieser Stelle die nächstbeste Rücksprungadresse vom Stack geholt und dorthin gesprungen. Dies ist zwar meist die Adresse 4, von wo aus der SEKA mit der Meldung '** Illegal Instruction at $000004' zurückkehrt, kann aber auch eine völlig andere sein, woraufhin sich der Rechner wieder in die Meditation begeben kann...

An den als Breakpoint eingegebenen Stellen setzt der SEKA einen ILLEGAL-Befehl, nachdem er den alten Inhalt dieser Speicherworte gerettet hat. Gelangt der Prozessor bei der Bearbeitung des Programms dann an einen solchen Befehl, so wird über den Illegal-Instruction-Vektor, welcher vom SEKA vorher entsprechend eingestellt wurde, in den Debugger gesprungen. Der SEKA repariert dann wieder die veränderten Speicherzellen und gibt dann die Statuszeile aus, an der man erkennen kann, wo das Programm abgebrochen wurde.

Diese Technik der Abbruch-Punkte ist sehr vorteilhaft, wenn man einen Fehler in einem Programm suchen will. Man kann dann z.B. vor eine zweifelhafte Routine einen Breakpoint setzen und das Programm starten. Wenn es dann an dieser Stelle wieder abbricht, kann man die Routine mit der Einzelschritt-Funktion 's' Schritt für Schritt durchlaufen und das Ergebnis jedes Befehls anhand der Statuszeile betrachten und so leicht dem Fehler auf die Spur kommen.

Diese Programmfehler, die sich so heimtückisch in Programme einschleichen, nennt man auch Bugs (Wanzen), weshalb auch ein Programm zum Entwanzen anderer Programme eben Debugger heißt.

4. Die ersten Programme

Jetzt ist es soweit, daß wir unsere neuen Kenntnisse anhand von Programmen erproben werden, die nicht unbedingt nur zur Demonstration dienen, sondern auch einen Zweck erfüllen. Dabei gehe ich davon aus, daß Sie den SEKA-Assembler besitzen und geladen haben, wobei in den Programmbeispielen die Zeilennummern weggelassen sind.

Bei der Verwendung eines anderen Assemblers müssen natürlich einige Dinge abweichend von den Vorgaben gehandhabt werden. Diese Unterschiede sind bereits im Kapitel über die verschiedenen Assembler besprochen worden.

Um die Beispielprogramme zum einen direkt ausprobieren und zum zweiten später weiterverwenden zu können, sind sie als Unterprogramme ausgeführt. Sie können daher nach der Assemblierung des Programms und ggf. der Zuweisung der gewünschten Werte in die entsprechenden Register mit 'j Programmname' das Programm starten und nachher das oder die Ergebnisse entweder direkt aus den Registerinhalten oder mit 'q Adresse' aus dem Speicher ablesen.

Beginnen wir nun mit einem einfachen Beispiel: der Addition von Zahlen aus einer Tabelle.

4.1 Addieren von Zahlen

Stellen Sie sich vor, Sie haben eine Reihe Zahlen im Speicher liegen, die Sie aufaddieren wollen. Nehmen wir als Beispiel 5 Zahlen in Wortbreite, deren Summe in das Register D0 geschrieben werden soll. Die einfachste Möglichkeit hierfür ist:

```
addieren1:
     clr.l   D0              ;D0 löschen (=0)
     move    tabelle,d0      ;ersten Eintrag in D0
     add     tabelle+2,d0    ;zweiten Eintrag addieren
     add     tabelle+4,d0    ;dritten Eintrag addieren
```

```
        add     tabelle+6,d0      ;vierten Eintrag addieren
        add     tabelle+8,d0      ;fünften Eintrag addieren
        rts                       ;zurück ins Hauptprogramm

tabelle:        dc.w  2,4,6,8,10
```

Probieren Sie doch einmal dieses Programm aus mit 'j addieren1'. Sie sehen, daß das Datenregister D0 tatsächlich die Summe der Werte beinhaltet.

Ich denke, Sie können sich schon denken, daß dies auch viel eleganter und kürzer realisierbar ist. Dafür eine kleine Motivation: Wie wäre es, wenn die Adresse der zu addierenden Daten variabel sein muß, d.h. wenn ein und dasselbe Programm verschiedene Tabellen addieren können soll?

Die obige Methode versagt dann natürlich, da sie nur ganz bestimmte Adressen ausliest und addiert. Doch wozu haben wir im Amiga einen Prozessor mit so vielen und vielseitigen Adressierungsarten?

Ich schlage also vor, daß wir dem Unterprogramm die Adresse der Tabelle in einem Adreßregister (z.B. A0) übergeben. Dieses Register kann dann als Zeiger auf die Tabelle direkt verwendet werden, und zwar mit Hilfe der indirekten Adressierung. Wir können also im Programm die Ausdrücke 'tabelle+x' in '2*x(a0)' umändern:

```
addieren1:
        clr.l   D0                ;D0 löschen (=0)
        move    0(a0),d0          ;ersten Eintrag in D0
        add     2(a0),d0          ;zweiten Eintrag addieren
        add     4(a0),d0          ;dritten Eintrag addieren
        add     6(a0),d0          ;vierten Eintrag addieren
        add     8(a0),d0          ;fünften Eintrag addieren
        rts                       ;zurück ins Hauptprogramm

tabelle:        dc.w  2,4,6,8,10
```

Auch dieses Programm addiert die 5 Zahlen ordnungsgemäß. Der Grund für die Schrittweite von 2 in den Angaben des Offsets liegt darin, daß wir ja auf Worte zugreifen, die je 2 Bytes groß sind.

Eine weitere 'Verschönerung' des Programms wäre es, anstelle von 'x(a0)' den Ausdruck '(a0)+' zu verwenden. Dabei wird nach jedem Zugriff auf die Tabelle das Adreßregister A0 automatisch hochgezählt, und zwar um die Anzahl Bytes, die auf einmal ausgelesen wurden (hier also um 2). Der Unterschied zu obiger Variante ist allerdings der, daß der Inhalt des Registers dabei verändert wird; dieser Zeiger zeigt dann auf das nächste unbenutzte Byte bzw. Wort im Speicher.

Nun kommt ein weiteres Handicap: die Anzahl der zu addierenden Worte soll auch variabel sein und z.B. im Register D1 übergeben werden. Nun müssen wir uns eine ganz andere Art der Programmierung einfallen lassen, da dies mit der alten Methode nicht realisierbar ist.

Es kommt jetzt der Begriff der Schleife ins Spiel. Wir müssen ja D1 mal das nächste Wort addieren. Als Adressierungsart können wir wieder (a0)+ verwenden (Adreßregister indirekt mit Postincrement), da uns dies automatisch den jeweils nächsten Wert liefert.

Nun zur Schleife. Diese wird so realisiert, daß nach jeder Addition der Inhalt des Zählers, in unserem Beispiel das Register D1, um 1 erniedrigt wird. Ist das Ergebnis dann Null, so ist die Arbeit beendet, andernfalls wird eine weitere Addition durchgeführt. Dies sieht als Programm dann so aus:

```
addieren2:
     clr.l   d0          ;D0 löschen
loop:                    ;Label für den Schleifenanfang
     add     (a0)+,d0    ;Wert aufaddieren
     subq    #1,d1       ;Zähler erniedrigen
     bne     loop        ;weitermachen, wenn nicht Null
     rts                 ;sonst fertig

tabelle:     dc.w  2,4,6,8,10
```

Setzen Sie nun bitte einmal mit 'x a0' den Zeiger auf die Adresse der Tabelle und mit 'x d1' den Zähler auf 5. Danach können Sie das Programm wieder mit 'j addieren2' starten. Dies können Sie dann auch einmal mit einem anderen Wert für den Zähler probieren (z.B. 3) und mit dem erwarteten Ergebnis vergleichen. Zufrieden?

Nun noch eine letzte Feinheit, um das Programm eleganter und auch schneller zu machen. Wir verwenden dazu den speziell für Schleifen vorgesehenen Befehl 'DBRA' des MC68000. Dieser Befehl erledigt nämlich zwei Schritte, für die wir in unserem Programm zwei Befehle verwendet haben: das Herunterzählen des Zählers und die bedingte Verzweigung. Diese Verzweigung findet allerdings solange statt, wie der Zähler positiv ist. Somit wird also auch bei 0 verzweigt, so daß unser Programm nicht mehr genauso wie vorher funktioniert. Wir müssen daher lediglich im Zähler die Anzahl der Werte-1 übergeben, hier also 4.

```
addieren2:
     clr.l   d0          ;D0 löschen
loop:                    ;Label für den Schleifenanfang
     add     (a0)+,d0    ;Wert aufaddieren
     dbra    d1,loop     ;D1 herunterzählen, verzweigen, wenn pos.
     rts                 ;sonst fertig

tabelle:     dc.w  2,4,6,8,10
```

Bei diesem Programm bietet es sich an, den Ablauf genau zu beobachten. Dafür laden wir mit 'x' wieder den Zeiger A0 und den Zähler D1 mit den nötigen Werten und stellen dann mit 'x pc' den Programmzähler auf 'addieren2'. Nun können Sie mit 's' jeweils einen Programmschritt ausführen lassen und das Ergebnis beobachten. Lassen Sie allerdings am Ende des Programms, wenn der Befehl RTS angezeigt wird, diesen nicht ausführen, da sonst eine Rücksprungadresse von Stapel geholt wird, die ja gar nicht daraufgelegt wurde!

Zum Abschluß dieses Beispiels noch eine kleine Aufgabe zum Selbermachen: Schreiben Sie doch das Programm einmal so um, daß es einige Bytes oder einige Langworte addiert! Und dann

versuchen Sie auch ein Programm zu schreiben, welches von dem ersten Wert in einer Tabelle die folgenden Werte alle subtrahiert, also bei der Tabelle

```
tabelle:  dc.w   50,8,4,6
```

den Wert 50-8-4-6, also 32 ($20) zurückgibt!

4.2 Sortieren einer Tabelle

Bleiben wir noch bei der Verarbeitung von Tabellen. Wir wollen nun aber nicht mehr nur die Daten dort auslesen, sondern auch verändern. Die Tabelle soll nämlich aufsteigend sortiert werden.

Überlegen wir erst einmal, wie wir das Sortieren überhaupt handhaben können. Die einfachste Methode dafür ist folgende:

Man vergleicht den ersten mit dem zweiten Wert. Ist der zweite Wert größer als der erste, so ist dies in Ordnung und man geht zum nächsten Schritt über, in dem man den zweiten mit dem dritten Wert vergleicht usw.. Ist man beim letzten Paar angekommen und jedesmal war der vorhergehende Wert kleiner als der nächste, so ist die Sortierung beendet (bzw. unnötig).

Findet man nun ein Wertepaar, in dem der zweite Wert kleiner ist als der erste, so werden diese beiden Werte einfach vertauscht. Damit dieses Ereignis nicht vergessen wird, setzt man irgendeinen Merker (hier vorzugsweise ein Register), welcher am Schluß der Tabelle überprüft wird. Ist er dann gesetzt, so ist die Tabelle wahrscheinlich noch nicht fertig sortiert: also den Merker wieder löschen und von vorne beginnen. Ist der Merker am Ende immer noch gelöscht, so ist die Sortierung abgeschlossen.

Bauen wir nun hierauf ein Programm auf. Zuerst bestimmen wir die nötigen Variablen und ordnen ihnen Register zu. Wir brauchen also einen Zeiger auf die zu sortierende Tabelle (A0), einen Zähler (D0) und einen Merker (D1). Da wir im Laufe des Programms immer wieder diese Werte verändern, richten wir noch

zusätzlich zwei Register ein, in denen wir die Anfangswerte (Adresse und Anzahl der Tabelleneinträge) zwischenspeichern: A1 und D2.

Fangen wir nun einmal an, das Programm aufzubauen. Wir übergeben dem Unterprogramm die Adresse der Tabelle in A1 und die Anzahl der Werte in D2.

```
sortieren:                  ;Einsprungadresse des Programms
     move.l  a1,a0          ;Zeiger mit Adresse laden
     move.l  d2,d0          ;Anzahl in den Zähler
     subq    #2,d0          ;Zählerwert korrigieren
     clr     d1             ;Merker löschen
```

Soweit die Vorbereitungen. Der Zeiger und der Zähler sind vorbereitet und der Merker gelöscht. Der Zähler ist um 2 verringert worden, da wir mit dem DBRA-Befehl arbeiten wollen (1 abziehen) und bei X Zahlen nur X-1 Vergleiche anstellen können (noch 1 abziehen).

Als nächstes folgt die Schleife, in der die Werte verglichen werden. Wir vergleichen dabei wortweise einen Wert mit dem nächsten. Das sieht dann so aus:

```
loop:
     move    2(a0),d3       ;nächsten Wert in Register D3
     cmp     (a0),d3        ;Werte vergleichen
```

Den Umweg über das Register D3 brauchen wir leider, da CMP (A0),2(A0) nicht möglich ist. Ist der zweite Wert größer als der erste oder gleich, können wir das Austauschen unterlassen, also:

```
     bcc     noswap         ;verzweigen, wenn größer oder gleich
```

Nun wird ggf. vertauscht (leider geht das nicht mit 'exc 2(a0),(a0)', da diese Adressierung nicht existiert):

```
swap:
     move    (a0),d1       ;ersten Wert retten
     move    2(a0),(a0)    ;zweiten in ersten Wert kopieren
     move    d1,2(a0)      ;und ersten in zweiten bringen
     moveq   #1,d1         ;Merker setzen
noswap:
```

Nun den Zeiger erhöhen und weiter mit dem nächsten Paar usw., bis der Zähler negativ wird:

```
     addq.l  #2,a0         ;Zeiger+2
     dbra    d0,loop       ;Schleife fortführen bis Ende
```

Jetzt schauen wir nach, ob der Merker gesetzt ist, und beginnen von vorne, wenn er nicht 0 ist:

```
     tst     d1            ;Merker testen
     bne     sortieren     ;noch nicht fertig sortiert!
     rts                   ;sonst fertig: zurückkehren...
```

Ist der Merker gelöscht, so sind wir fertig und das Unterprogramm beendet. Es wird dann mit RTS ins Hauptprogramm zurückgekehrt.

Und nun das Programm noch einmal auf einen Blick:

```
sortieren:                 ;Einsprungadresse des Programms
     move.l  a1,a0         ;Zeiger mit Adresse laden
     move.l  d2,d0         ;Anzahl in den Zähler
     subq    #2,d0         ;Zählerwert korrigieren
     clr     d1            ;Merker löschen
loop:
     move    2(a0),d3      ;nächsten Wert in Register D3
     cmp     (a0),d3       ;Werte vergleichen
     bcc     noswap        ;verzweigen, wenn größer oder gleich
swap:
     move    (a0),d1       ;ersten Wert retten
     move    2(a0),(a0)    ;zweiten in ersten Wert kopieren
     move    d1,2(a0)      ;und ersten in zweiten bringen
     moveq   #1,d1         ;Merker setzen
```

```
noswap:
     addq.l  #2,a0        ;Zeiger+2
     dbra    d0,loop      ;Schleife fortführen bis Ende
     tst     d1           ;Merker testen
     bne     sortieren    ;noch nicht fertig sortiert!
     rts                  ;sonst fertig: zurückkehren...
```

Hier sind also einige Register verwendet worden, um aktuelle Werte zu speichern. Es kommt jedoch oft vor, daß ein Unterprogramm keine oder nur bestimmte Register verändern darf. Zu diesem Zweck gibt es einen Maschinenbefehl, der auf einen Schlag mehrere Register auf den Stapel legen und damit retten kann. Gemeint ist der Befehl MOVEM, was soviel wie 'Move Multiple' heißt. Fügen wir diesen Befehl einmal in unser Programm ein, natürlich zweimal, da die geretteten Werte auch wieder vom Stapel geholt werden. Dafür brauchen wir allerdings ein weiteres Label, welches wir 'start' nennen wollen und wo das Unterprogramm dann gestartet wird:

```
start:
     movem.l d0-d7/a0-a6,-(sp)      ;Register retten
sortieren:
     usw...
     ...
     ...
     bne     sortieren              ;noch nicht fertig
     movem.l (sp)+,d0-d7/a0-a6      ;Register wiederholen
     rts                            ;und fertig!
```

Dieser mächtige Befehl verschiebt also mehrere Register auf einmal. Dabei kann man angeben, welche Register verschoben werden sollen. Will man z.B. nur die Register D1,D2,D3,D7,A2 und A3 verschieben, so schreibt man

```
     movem.l d1-d3/d7/a2-a3,-(sp)
```

Bevor wir nun das Thema Sortierung beenden, wieder eine kleine Aufgabe: Versuchen Sie bitte einmal das Programm so zu verändern, daß es umgekehrt sortiert, also die größte Zahl nach vorne und die kleinste Zahl nach hinten bringt!

4.3 Zahlensysteme umwandeln

Wie Sie ja im Kapitel über die Zahlensysteme gemerkt haben, ist die Umrechnung von Zahlen in ein anderes Zahlensystem teilweise recht aufwendig. Außerdem gibt es ja auch eine weitere Art der Umwandlung von Zahlen: von oder in Zeichenketten, welche von Tastatur eingegeben oder am Bildschirm ausgegeben werden können!

Wir wollen nun aus dieser großen Menge von möglichen Umwandlungen einige herausnehmen und als Programme formulieren. Beginnen wir bei der Umwandlung von Binärzahlen in eine Zeichenkette, die den Wert der Zahl sedezimal ausdrückt.

4.3.1 Binär- in Sedezimalzahlen

Zuerst müssen die Anfangs- und Endbedingungen überlegt werden. Legen wir sie als Beispiel so fest, daß der Inhalt des Datenregisters D1 8-stellig, also als Langwort, als ASCII-Zeichen in einen bestimmten Speicherbereich gelegt werden soll, von wo aus sie evtl. später ausgegeben werden kann.

Der Vorteil der Sedezimaldarstellung gegenüber z.B. der Dezimalschreibweise zeigt sich gerade bei dieser Problemstellung sehr deutlich. Um eine sedezimale Ziffer aus einem Speicherwort heraus zu ermitteln, brauchen wir nur die entsprechenden 4 Bit, also das Halbbyte herauszunehmen und zu verarbeiten. Ein jedes dieser Halbbytes, auch Nibble genannt, enthält ja genau eine Ziffer der Sedezimalzahl!

Betrachten wir nun diese Verarbeitung eines Halbbytes, von dem wir annehmen, daß es in D2 liegt. Um aus diesem ein druckbares Zeichen zu erhalten, muß es in ein Zeichen des ASCII-Codes umgewandelt werden. Diese Codes sind für die möglichen 16 Zeichen einer Sedezimalziffer:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$30	$31	$32	$33	$34	$35	$36	$37	$38	$39	$41	$42	$43	$44	$45	$46

Um also die Ziffern 0-9 umzuwandeln, braucht man nur $30 zu addieren. Bei den Buchstaben A-F, weiche ja den Wert 10-15 haben, muß dagegen $37 addiert werden. Unser Programm zur Auswertung eines Halbbytes muß daher zuerst die Unterscheidung treffen, ob es sich bei dem Wert des Halbbytes um einen Wert zwischen 0 und 9 oder zwischen A und F handelt, und gemäß dieser Unterscheidung entweder $30 oder $37 addieren.

Setzen wir dies einmal in ein Maschinenprogramm um, und zwar als Unterroutine, da diese für jede Stelle des geplanten Langwortes aufgerufen werden muß.

```
nibble:
        and     #$0f,d2     ;unteres Nibble ausblenden
        add     #$30,d2     ;schon mal $30 addieren
        cmp     #$3a,d2     ;war's eine Ziffer?
        bcs     ok          ;ja: fertig
        add     #7,d2       ;sonst noch 7 addieren
ok:
        rts                 ;fertig
```

Diese Routine wandelt also das in D2 enthaltene Nibble in ein ASCII-Zeichen um, welches dem sedezimalen Wert des Nibbles entspricht. Um ein ganzes Byte umzuwandeln, müssen wir diese Routine zweimal aufrufen. Hier ein Programm dafür, welches als Parameter in A0 die Adresse eines Puffers zum Ablegen der Zeichen und in D1 das umzuwandelnde Byte übergeben bekommt:

```
        ...
        lea     puffer,a0   ;Zeiger auf Puffer
        move    #$4a,d1     ;umzuwandelndes Byte (Beispiel)
        bsr     byte        ;und umwandeln
        ...

byte:
        move    d1,d2       ;Wert in D2 bringen
        lsr     #4,d2       ;oberes in unteres Nibble bringen
        bsr     nibble      ;D2 umwandeln
        move.b  d2,(a0)+    ;Zeichen in Puffer legen
```

```
        move    d1,d2       ;Wert in D2
        bsr     nibble      ;unteres Nibble umwandeln
        move.b  d2,(a0)+    ;und in Puffer bringen
        rts                 ;fertig

nibble:
        ...                 ;Programm siehe oben

puffer:   blk.b 9,0         ;Puffer für Langwort-Daten
```

Zuerst wird der umzuwandelnde Wert in D2 gebracht. Danach wird dieses Register 4mal nach rechts geschoben, um das obere Nibble in die unteren 4 Bits zu befördern. Dadurch wird nach dem nun folgenden Unterprogramm-Aufruf das HI-Nibble zuerst mit 'move.b d2,(a0)+' in den Puffer gebracht. Danach wird das Originalbyte wieder in D2 gebracht und umgewandelt, was schließlich das LO-Nibble als ASCII-Zeichen in D2 ergibt, welches in das nächste Byte des Puffers gebracht wird.

Dieser Puffer ist bereits lang genug, um ein Langwort in Ziffern aufzunehmen, inklusive eines abschließenden Null-Bytes, welches üblicherweise von Bildschirmausgabe-Routinen als Textende-Zeichen benötigt wird. Die Bildschirmausgabe verschieben wir aber noch auf später, kümmern wir uns zuerst einmal um die vollständige Umwandlung des Langwortes.

Auch bei der Umwandlung des gesamten Langwortes müssen wir darauf achten, daß wir die Reihenfolge der Nibbles einhalten. Um also das erste Mal die 'nibble'-Routine aufrufen zu können, müssen wir das obere der 8 Nibbles des Langwortes in die unteren 4 Bits bringen, natürlich ohne daß dabei etwas verlorengeht.

Hierfür eignet sich der Verschiebe-Befehl LSR nicht so gut, da bei diesem Befehl die herausgeschobenen Bits verlorengehen. Besser geeignet sind hier die Rotations-Befehle wie ROR oder ROL, die das herausgeschobene Bit auf der anderen Seite wieder hineinschieben.

Wenn wir also das Original-Langwort in D1 4 mal nach links schieben, werden die oberen 4 Bits in die unteren 4 Bits gescho-

ben, von wo aus wir sie mit unserer 'nibble'-Routine auswerten und dann das entstandene ASCII-Zeichen in den Puffer bringen können. Das ganze 8mal, und das Langwort ist vollständig umgewandelt, sogar mit dem unerhörten Luxus, das das Register D1 nachher wieder den ursprünglichen Wert hat!

```
hexlong:
     lea      puffer,a0    ;Zeiger auf den Puffer
     move     #7,d3        ;Zähler der Nibbles auf 8-1
loop:
     rol      #4,d1        ;oberes Nibble nach unten bringen,
     move     d1,d2        ;in D2 schreiben
     bsr      nibble       ;und umwandeln
     move.b   d2,(a0)+     ;Zeichen in Puffer
     dbra     d3,loop      ;das ganze 8mal
     rts                   ;fertig !

nibble:
     ...                   ;Programm siehe oben

puffer:   blk.b 9,0        ;Puffer für Langwort-Daten
```

Geben Sie dieses Programm mit der Nibble-Routine bitte nun in den SEKA ein und assemblieren es. Danach probieren Sie es bitte mit einem Wert in D1 von $12345678 aus. Sie werden feststellen, daß da ein Fehler im Programm sein muß, da im Puffer die Ziffern '56785678' anstatt '12345678' stehen. Versuchen Sie doch einmal, diesen Fehler zu finden!

Na, haben Sie ihn gefunden? Dieser Fehler ist ein sehr häufig vorkommender Flüchtigkeitsfehler, der sich jedoch nicht so leicht erkennen läßt. Durch die Annahme des Assemblers, daß Sie die Rotieroperation in Wortbreite vornehmen wollen, da die Angabe '.l' fehlt, wird immer nur das untere Wort von D1 gerollt, so daß Sie zweimal dieselben Werte geliefert bekommen. Ändern Sie jedoch die Anweisung in 'rol.l' ab, so arbeitet das Programm ordnungsgemäß.

Durch diesen Fehler haben Sie aber auch erkannt, wie einfach es ist, mit obigem Programm 4stellige Sedezimalzahlen in ASCII-Zeichen umzuwandeln. Sie brauchen dafür nur das '.l' wieder

aus der 'rol'-Anweisung herauszunehmen und den Zähler von 7 auf 3 zu ändern: schon erhalten Sie ein 4stelliges Sedezimalwort.

Auch hierzu eine kleine Aufgabe: Ändern Sie doch das Programm dahingehend, daß 6stellige Sedezimalzahlen herauskommen (D1 muß nicht unbedingt unverändert bleiben...)!

Wenden wir uns nun einem weiteren Umwandlungsproblem zu: der Wandlung in eine 4stellige Dezimalzahl.

4.3.2 Binär- in Dezimalzahlen

Wie man leicht erkennt, ist dies nicht ganz so einfach wie die Umwandlung in eine Sedezimalzahl. Hierbei kann man nämlich nicht so einfach die einzelnen Ziffern durch Auswertung von Bitgruppen erhalten. Wir müssen uns daher eine andere Methode überlegen.

Sehen wir uns dafür einmal an, wie eine Dezimalzahl eigentlich aufgebaut ist. Bei 4 Stellen bedeutet die vorderste Stelle der Zahl, wie oft in dem Gesamtwert die 1000 enthalten ist, die nächste Stelle gibt an, wie oft die 100 in dem Rest enthalten ist usw.

Wenn wir also nun einen Wert in einem Register haben und dieses Register durch 1000 teilen, erhalten wir im Ergebnis den Wert der vordersten Stelle unserer Dezimalzahl. Da der für die Division zuständige Maschinenbefehl DIV uns neben dem ganzzahligen Ergebnis der Teilung auch den Rest liefert, können wir mit diesem Rest sehr einfach weiterarbeiten. Wir teilen nämlich diesen Rest durch 100, erhalten damit die 100er-Stelle der Dezimalzahl, teilen den Rest wiederum durch 10 und erhalten so schließlich die 10er-Stelle und mit dem Rest auch die Einerstelle der gesuchten Zahl.

Auf diese Weise ist es gar nicht so schwer wie zunächst zu vermuten war. Hier das Programm, welches der Reihe nach die oben beschriebenen Schritte ausführt und den Wert von D1 in den Puffer bringt:

```
dezi_4:     ;Unterprogramm für 4stellige Dezimalzahlen
     lea    puffer,a0   ;Zeiger auf den Puffer
     divu   #1000,d1    ;Teilung durch 1000
     bsr    ziffer      ;Ergebnis auswerten und Rest runter
     divu   #100,d1     ;Teilung durch 100
     bsr    ziffer      ;Ergebnis auswerten und Rest runter
     divu   #10,d1      ;Teilung durch 10
     bsr    ziffer      ;Ergebnis auswerten und Rest runter
                        ;und den Rest direkt auswerten

ziffer:
     add    #$30,d1     ;Ergebnis in ASCII umwandeln
     move.b d1,(a0)+    ;und in Puffer bringen
     clr    d1          ;unteres Wort löschen
     swap   d1          ;und Rest nach unten bringen
     rts                ;zurück

puffer:  blk.b  5,0     ;Puffer für die ASCII-Zeichen
```

An diesem Programm sehen Sie einen kleinen Trick, der öfters angewandt wird: die Unterroutine 'ziffer', welche dreimal von der Unterroutine 'dezi_4' aus aufgerufen wird, wird dann plötzlich zu einem Teil der 'dezi_4'-Routine. Nach dem dritten Aufruf von 'ziffer' wird nämlich in 'ziffer' direkt weitergearbeitet, ohne daß ein BSR oder JSR stattgefunden hat. Stößt der Prozessor dann auf das RTS, so wird aus der Unterroutine zurückgekehrt, diesmal aber nicht in das 'dezi_4'-Unterprogramm, sondern in das Hauptprogramm! Auf diese Weise spart man das vierte 'bsr ziffer' und das 'rts' für die Beendigung der 'dezi_4'-Routine.

Probieren Sie dieses Programm nun einmal aus. Dabei sollten Sie aber auch beachten, daß Sie als umzuwandelnden Wert eine Zahl in D1 schreiben, die wirklich kleiner als 9999 ist, da sonst sehr merkwürdige Ergebnisse entstehen können...

Und nun begeben wir uns an die Umkehrung der oben vorgestellten Umwandlungen: Zeichenketten in Binärzahlen.

4.3.3 Sedezimal- in Binärzahlen

Wenn man eine Zeichenkette, welche eine Sedezimalzahl darstellt, genau betrachtet, sieht man, daß wieder jedes Zeichen ein Halbbyte darstellt. Man muß also ein Programm schreiben, was genau andersherum arbeitet wie jenes, das eine Binärzahl in eine Sedezimalzahl umwandelt.

Man kann dabei zwei verschiedene Ansätze machen:

1. die Anzahl der Sedezimalziffern ist bekannt oder
2. die Anzahl ist unbekannt.

Der erste Fall ist natürlich leichter zu programmieren, hat allerdings den Nachteil, daß man beispielsweise für die Eingabe einer vierstelligen Sedezimalzahl auch für den Wert 1 vier Ziffern, also 0001, schreiben muß. Dies ist nicht gerade komfortabel, so daß wir direkt den zweiten Fall wählen.

Beginnen wir mit der Umwandlung einer einzelnen Ziffer. Angenommen, wir übergeben einen Zeiger auf diese Ziffer im Adreßregister A0 an ein Unterprogramm zur Umwandlung und wollen den erhaltenen Binärwert im Datenregister D0 zurückbekommen.

Das Programm sieht dann etwa so aus:

```
nibblein:                ;* Umwandlung des Nibbles aus (A0)
     clr.l   d0          ;D0 löschen
     move.b  (a0)+,d0    ;Ziffer holen,A0 erhöhen
     sub     #'A',d0     ;$41 subtrahieren
     bcc     ischar      ;kein Übertrag: es war A-F
     add     #7,d0       ;sonst Wert korrigieren
ischar:
     add     #10,d0      ;Wert korrigieren
     rts
```

Verfolgen wir den Ablauf dieses Programms an Hand eines Beispiels. Nehmen wir an, in der durch A0 angezeigten Spei-

cherstelle stehe das Zeichen 'B', welches durch den ASCII-Wert $42 repräsentiert wird. Diese Zahl wird zuerst einmal in D0 geladen, nachdem dieses Register gelöscht wurde.

Durch die Subtraktion des Wertes $41 wird daraus der Wert $1, womit wir bereits fast fertig sind. Vor der Rückkehr ins Hauptprogramm wird nämlich noch einmal 10 addiert, wodurch wir auf den richtigen Wert 11, also $B, kommen.

Hätten wir eine Ziffer übergeben, so wäre bei der Subtraktion ein Überlauf ins Negative aufgetreten und damit das C-Flag gesetzt worden. Nehmen wir als zweites Beispiel die Ziffer 5.

Der ASCII-Wert der 5 ist $35. Bei der Subtraktion wird davon $41 abgezogen, was -12 ergibt und das C-Flag setzt. Ist dies der Fall, so wird beim BCC-Befehl nicht verzweigt, sondern danach der Wert 7 addiert, wodurch wir auf -5 kommen. Dann wird die 10 addiert und wir erhalten die 5: fertig!

Diese Routine hat allerdings einen Nachteil. Wird ein falsches Zeichen übergeben, welches überhaupt keine Sedezimalziffer darstellt, so wird irgendein unsinniges Ergebnis zurückgegeben. Aber lassen wir Fehleingaben erst einmal außer acht.

Gehen wir nun weiter zu mehrstelligen Sedezimalzahlen. Die erste Ziffer, die wir umwandeln, hat den höchsten Wert und muß daher auch das höchste Nibble darstellen. Um dies zu realisieren und dennoch mit beliebig langen Zahlen hantieren zu können (nicht ganz beliebig, die Zahl sollte in ein Langwort passen, also maximal 8-stellig sein), wenden wir einen Trick an.

Betrachten wir hierzu gleich das ganze Programm, welches die ganze Umrechnung erledigt und das Ergebnis in D1 zurückgibt. Es wird dabei davon ausgegangen, daß der Zeiger auf die Zeichenkette in A0 übergeben wird und diese Zeichenkette mit einem Nullbyte abgeschlossen ist.

```
hexin:                  ;* Umwandlung einer Sedezimalzahl
     clr.l   d1         ;D1 erst löschen
hexinloop:
     tst.b   (a0)       ;Ziffer testen
     beq     hexinok    ;Null, also fertig
     bsr     nibblein   ;Ziffer umwandeln
     lsl.l   #4,d1      ;Ergebnis verschieben
     or.b    d0,d1      ;Nibble einfügen
     bra     hexinloop  ;und weitermachen
hexinok:
     rts
```

Der erwähnte Trick liegt darin, daß bei Bedarf das Ergebnis viermal, also um ein Nibble, nach links geschoben wird. Dadurch wird der Stellenwert der letzten Ziffer um erhöht und Platz geschaffen für das Nibble, welches durch die 'nibblein'-Routine geliefert wird. Stößt das Programm auf das Nullbyte am Ende der Zeichenkette, so wird dies durch den TST.B-Befehl festgestellt und das Programm beendet. Das Ergebnis liegt nun im Langwort D1!

Um nun auch eventuelle Fehleingaben abzufangen, müssen wir einige Änderungen in diesem Programm vornehmen. Wir setzen dafür an der Stelle an, wenn wir von der 'nibblein'-Routine den angenommenen Wert des aktuellen Zeichens zurückbekommen.

Wenn der Wert in D0 größer ist als $F, so ist ein Fehler aufgetreten. Dies können wir auf mehrere Arten feststellen. Wir wählen den einfachen Weg, indem wir den Wert mittels einer 'CMP #$10,D0'-Operation mit $10 vergleichen. Ist er kleiner, so ist ja dann das C-Flag gesetzt, da bei der Subtraktion, welche CMP ausführt, ein Übertrag stattgefunden hat. Andernfalls ist C gelöscht, was auf einen Fehler hindeutet.

Wir können durch diesen Trick übrigens auch den Test auf das Nullbyte wegfallen lassen, da auch dieses ein ungültiges Zeichen darstellt. Das gesamte Programm sieht somit folgendermaßen aus:

```
hexin:                  ;* Umwandlung einer Sedezimalzahl
     clr.l   d1         ;D1 erst löschen
hexinloop:
     bsr     nibblein   ;Ziffer umwandeln
     cmp     #$10,d0    ;Test, ob gültig
     bcc     hexinok    ;nein, also fertig
     lsl.l   #4,d1      ;Ergebnis verschieben
     or.b    d0,d1      ;Nibble einfügen
     bra     hexinloop  ;und weitermachen
hexinok:
     rts                ;Ende der Umwandlung

nibblein:               ;* Umwandlung des Nibbles aus (A0)
     clr.l   d0         ;D0 löschen
     move.b  (a0)+,d0   ;Ziffer holen,A0 erhöhen
     sub     #'A',d0    ;$41 subtrahieren
     bcc     ischar     ;kein Übertrag: es war A-F
     add     #7,d0      ;sonst Wert korrigieren
ischar:
     add     #10,d0     ;Wert korrigieren
     rts
```

Dies war also die Methode, Sedezimal- in Binärzahlen umzuwandeln. Gehen wir nun zu Dezimalzahlen über, was in dieser Umwandlungsrichtung kaum komplizierter ist.

4.3.4 Dezimal- in Binärzahlen

Im großen und ganzen können wir die gleiche Methode wie bei Sedezimalzahlen anwenden. Da wir auch hier von einer unbekannten Anzahl Ziffern ausgehen, ist auch die Technik der Erhöhung des momentanen Stellenwertes ähnlich. Wir können dies zwar nicht durch einfache Verschiebung erreichen, sondern müssen das Ergebnis mit 10 multiplizieren und den Wert der Ziffer addieren.

Hier das gesamte Programm für die Umwandlung von Dezimalzahlen:

```
dezin:                  ;* Umwandlung einer Dezimalzahl
     clr.l   d1         ;D1 erst löschen
dezinloop:
     bsr     zifferin   ;Ziffer umwandeln
     cmp     #10,d0     ;Test, ob gültig
     bcc     dezinok    ;nein, also fertig
     mulu    #10,d1     ;Ergebnis verschieben
     add     d0,d1      ;Nibble einfügen
     bra     dezinloop  ;und weitermachen
dezinok:
     rts                ;Ende der Umwandlung

zifferin:               ;* Umwandlung des Nibbles aus (A0)
     clr.l   d0         ;D0 löschen
     move.b  (a0)+,d0   ;Ziffer holen,A0 erhöhen
     sub     #'0',d0    ;$30 subtrahieren
     rts
```

Dieses Programm kann leider 'nur' Zahlen bis 655350 umwandeln, im Gegensatz zu der Sedezimalzahlen-Umwandlungsroutine. Dies liegt daran, daß der MULU-Befehl nur 16-Bit-Worte multiplizieren kann. Die letzte Multiplikation, die zum Erreichen des größten Wertes korrekt erfolgen würde, wäre die von $FFFF*10, also 65535*10, was den Wert 655350 ergäbe. Normalerweise reicht dieser Zahlenbereich jedoch aus, so daß wir uns die Verkomplizierung des Programms sparen können.

5. Hardware-Register-Anwendung

Es gibt noch einen anderen Weg außer den Bibliotheksfunktionen, um an einige Informationen heranzukommen oder bestimmte Hardware-Funktionen auszulösen: die Hardware-Register. Dies sind Speicherzellen, die an bestimmten Adressen stehen, welche weder im RAM noch im ROM liegen. Es handelt sich dabei nämlich um die direkten Schnittstellen zwischen dem Prozessor und seinen Peripheriebausteinen.

So findet man für jeden dieser Bausteine eine ganze Reihe Hardware-Register, durch die auch der Prozessor zugreift und damit Graphik, Sound und Ein-/Ausgaben allgemein steuert. Hier bietet sich gerade für den Assembler-Programmierer ein weites Betätigungsfeld, wovon hier nur einige Beispiele angeführt werden sollen.

Die Register werden im allgemeinen byteweise verwendet. Ein Beispiel hierfür finden Sie im nächsten Kapitel.

5.1 Sondertasten abfragen

Laden Sie bitte den SEKA-Assembler oder ein ähnliches Monitor-Programm, und geben Sie ein:

```
'q $bfec00'
```

Sie erhalten die byteweise Auflistung der Adressen $BFEC00 bis $BFEC7F, in denen sich zwei Bytes immer wiederholen. Diese beiden Bytes stellen immer wieder den Zustand zweier Hardware-Register dar.

Diese Spiegelung liegt daran, daß bei der Decodierung der Adresse durch den angesprochenen Baustein nicht alle Adreßbits verwendet werden. Dadurch spielen für die Adressierung dieser Register nur die oberen 2 Bytes der Adresse sowie das niedrigste Bit, Bit 0, eine Rolle. Die Adresse der beiden Register lautet somit etwa so: $BFECxx, wobei das untere Adreßbyte xx nur im

Bit 0 die Information über das gewünschte Register enthält, die Bits 1-7 dagegen egal sind. Diese eigenartige Adressierung findet sich bei den meisten Hardware-Registern wieder.

Doch betrachten wir nun einmal, welche Information in diesen Registern enthalten ist. Beachten wir dafür das zweite Register, $BFEC01, welches $77 enthält. Geben Sie nun noch einmal

```
'q $bfec00'
```

ein und drücken unmittelbar nach der Betätigung der Return-Taste die Alternate-Taste. Sie werden feststellen, daß sich der Inhalt jedes zweiten Bytes ($BFEC01, $BFEC03 usw.) in $37 verwandelt. Dies gibt den Zustand der gedrückten Sondertasten an. Ebenso gilt dies für die anderen Sondertasten, und zwar ergeben folgende Tasten die Bytes:

Shift links	$3F
Shift rechts	$3D
Control	$39
Alternate	$37
Amiga links	$33
Amiga rechts	$31

Mit diesem Register kann man also aus einem laufenden Maschinenprogramm ständig testen, ob eine dieser Tasten gedrückt wurde, und dann ggf. irgendwelche Funktionen auslösen oder beenden. Ein Programmteil sähe dann etwa so aus:

```
stasten = $bfec01

    ...
    cmp.b     #$37,stasten    ;Alternate gedrückt?
    beq       funktion1       ;ja!
    cmp.b     #$31,stasten    ;oder Amiga-Taste rechts?
    beq       funktion2       ;ja!
    ...                       ;weder noch...
```

5.2 Laufende Zeit ermitteln

Der Amiga verfügt über eine eingebaute Uhr, die die Uhrzeit und das Datum enthält. Diese Uhr kann man mit Hilfe einer Bibliotheksfunktion auslesen. Diese Zeitangabe ist jedoch nicht für jedes Programm sinnvoll.

Will man in einem Programm feststellen, wieviel Zeit zwischen zwei Ereignissen vergangen ist, wird dies bei Verwendung der Uhrzeit recht ungenau. Interessanter wäre eine laufende Zeit, die in einem Hardware-Register läge und recht schnell, aber genau hochzählt. Einen solchen Zeitgeber enthält auch einer der Bausteine des Amiga: der Ein-/Ausgabe-Portbaustein. In einem dieser Chips läuft ein 24 Bit breiter Zähler ständig mit einem Takt von 50 Hertz (in amerikanischen Geräten mit 60 Hertz).

Diese 24 Bits können allerdings nicht auf einmal, etwa mit einem MOVE.L-Befehl, ausgelesen werden, da auch dieses Register auf 3 Bytes verteilt ist. Das niederwertigste der drei Bytes liegt an der Adresse $$BFE801, das mittlere bei $BFE901 und das höchste mit den Bits 16-23 in $BFEA01.

Ein Beispiel für die Anwendung dieser Register: Feststellen der Laufzeit einer Unterroutine.

```
        ...
        bsr     gettime         ;momentane Zeit in D7
        move.l  d7,d6           ;und in D6 retten
        bsr     routine         ;zu testende Routine starten
        bsr     gettime         ;nochmal die Zeit holen
        sub.l   d6,d7           ;verstrichene Zeit liegt in
        ...                     ;1/50 Sekunden in D7!
        ...

gettime:
        move.b  $bfea01,d7      ;HI-Byte in D0
        lsl.l   #4,d7           ;zweimal um 4,
        lsl.l   #4,d7           ;also um 8 Bits verschieben
        move.b  $bfe901,d7      ;MID-Byte dazunehmen
```

```
lsl.l    #4,d7
lsl.l    #4,d7          ;wieder weiterschieben
move.b   $bfe801,d7     ;und noch das LO-Byte dazu
rts                     ;fertig
```

5.3 Maus und Joystick auslesen

Auch für Maus und Joystick gibt es zwei Hardware-Register, die den Zustand bzw. die Position dieser Eingabegeräte anzeigen. Interessant dabei ist, daß ein und derselbe Port sowohl mit Maus als auch mit Joystick funktioniert, obwohl die Funktion der beiden völlig unterschiedlich ist.

Der Joystick besitzt 4 Schalter, die bei einer Bewegung geschlossen werden und Massepotential (-) auf den oder die entsprechenden Eingänge des Joystick-/Maus-Port legt. Die Maus dagegen gibt bei einer Bewegung viele und schnelle Signale aus, zwei für horizontal und zwei für vertikal.

Der Rechner muß also die Ports ständig überwachen und bei einer Bewegung die Signale auswerten und die neue Mausposition berechnen. Dies ist aber glücklicherweise nicht die Aufgabe des Prozessors, da dieser dabei zu viel zu tun hätte.

Wir finden den Status der Maus-/Joystick-Ports an den Adressen $DFF00A für den vorderen Port (1) und $DFF00C für den hinteren Port (2). Die Information dieser Worte ist für die vertikale Mausbewegung im unteren Byte und für die horizontale Bewegung im oberen Byte enthalten.

Doch Vorsicht: Diese Adressen dürfen nicht mit dem SEKA ausgelesen werden! Das führt nämlich aus unerfindlichen Gründen zum Absturz des Rechners, was zwar interessant aussieht (das Bild beginnt zu tanzen), aber nur mit Reset und damit totalem Datenverlust zu beheben ist!

Um diese Register auszulesen, schreiben wir also ein kleines Programm:

```
joy  =  $dff00a

run:
     move   joy,d6        ;Daten 1 in D6
     move   joy+2,d7      ;Daten 2 in D7
     rts
```

Wenn Sie dieses Programm assemblieren und mit 'j run' starten, erhalten Sie in D6 und D7 die Inhalte der beiden Register. Bewegen Sie nun die Maus ein wenig und starten das Programm erneut.

Wie Sie sehen, hat sich der Wert in D6 verändert. Verschieben Sie die Maus nur horizontal, so ändert sich auch nur der Wert des unteren Bytes, bei vertikaler Bewegung das obere Byte.

Was Sie so erhalten haben, ist allerdings nicht die absolute Position des Mauszeigers auf dem Bildschirm. Dies können Sie leicht feststellen, wenn Sie die Maus nach ganz links oben schieben, den Wert mit 'j run' auslesen und die Maus dann weiter nach links schieben. Der Mauszeiger bewegt sich dann natürlich nicht mehr, der Wert in dem Register dagegen schon. Sie sehen, daß der Inhalt dieser Register immer relativ zu sehen ist.

Ändern wir nun das Programm folgendermaßen:

```
joy  =  $dff00a

run:
     move   d7,d6         ;alte Position in D6
     move   joy,d7        ;neue Position in D7
     sub    d7,d6         ;Differenz in D6
     rts
```

Starten Sie dieses Programm nun zweimal. Das Ergebnis in D6 ist 0. Wenn Sie jetzt die Maus bewegen und nochmal starten, erhalten Sie in D6 nach jedem Start die Differenz der alten und neuen Position aus dem Register, also den vertikalen und horizontalen Abstand der Mausposition jetzt zu derjenigen des letzten Versuches. Auf diese Weise können Sie mit diesem Register die relative Mausbewegung zwischen zwei Abfragen ermitteln.

Nun zu der Abfrage des Joysticks. Stecken Sie dafür einen Joystick in Port 2 und ändern im Programm die Adresse $DFF00A in $DFF00C um. Starten Sie dieses Programm wieder zweimal, so erhalten Sie wieder eine Null in D6.

Bewegen Sie den Joystick jetzt nach oben und starten das Programm, so erhalten Sie als Ergebnis den Wert $FF00. Es wurde also vom oberen Byte 1 abgezogen. Lassen Sie nun den Joystick los und starten erneut, finden Sie in D6 den Wert $100: es wurde 1 addiert. Den selben Effekt hat die Bewegung des Joysticks nach links, nach dem Loslassen wird wieder 1 subtrahiert.

Die einzelnen Bewegungen und deren Ergebnis bei obigem Programm sind:

```
hoch    $FF00, HI-Byte -1
runter  $FFFF, LO-Byte -1
links   $0100, HI-Byte +1
rechts  $0001, LO-Byte +1
```

Sehr zuverlässig sind diese Werte allerdings nicht. Wenn Sie nämlich einmal den Joystick so richtig 'umrühren' und sich das Ergebnis ansehen, finden Sie einen unsinnigen Wert in D6. Dies liegt einfach daran, daß die Hardware von einer angeschlossenen Maus ausgeht. Dennoch ist diese Methode die schnellste, um einen Joystick auszulesen. Somit kann hier auch evtl. ein externes Gerät angeschlossen werden, welches auswertbare TTL-Signale an den Port anlegt und so durch ein Maschinenprogramm überwacht werden kann.

Was uns nun noch für die Auswertung des Joysticks fehlt, ist die Abfrage des Feuerknopfes. Der Zustand dieses Knopfes ist in Bit 7 des Bytes enthalten, welches Sie in der Speicherzelle $BFE001 finden. Ist dieses Bit gesetzt, so ist der Knopf nicht gedrückt. Dies gilt für einen an Port 2 angeschlossenen Joystick. Bit 6 dieses Bytes zeigt dagegen den Zustand des Knopfes an, wenn der Joystick in Port 1 sitzt, oder des linken Mausknopfes.

Bleiben wir bei Port 2. Will man Bit 7 des Bytes testen, um bei gedrücktem Knopf irgendeine Funktion auszulösen, so ist dies

sehr einfach. Bit 7 gibt ja das Vorzeichen eines Bytes an. Dies können wir uns mit folgendem Programmstück zunutze machen:

```
tst.b   $bfe001     ;Feuerknopf 2 gedrückt?
bpl     feuer       ;ja! Verzweigen
```

Der TST.B-Befehl testet das adressierte Byte und setzt ggf. das Z- und das N-Flag. Ist das N-Flag gesetzt, so wissen wir, daß Bit 7 des getesteten Bytes gesetzt ist. Da der Feuerknopf LO-Potential schaltet, wird das Bit bei gedrücktem Knopf gelöscht und das N-Flag durch den TST-Befehl auch. Der BPL-Befehl des obigen Programms verzweigt somit, wenn der Knopf gedrückt ist, da PL für Plus steht und bei gelöschtem Vorzeichenbit erfüllt ist.

5.4 Bedienung der Parallelschnittstelle

Eine noch wesentlich interessantere Möglichkeit, externe Geräte an den Amiga anzuschließen, bietet der Parallelport an der Rückseite des Gerätes, an welchem normalerweise der Drucker angeschlossen ist. Diese Buchse bietet nämlich mehrere Datenleitungen, mit denen man wahlweise Signale sowohl einlesen als auch ausgeben kann.

Intern liegt an dieser Schnittstelle ein Baustein, welcher PIA (Parallel Interface Adapter) genannt wird. Dieser Baustein wurde speziell für parallele Datenübertragung entwickelt. An die Schnittstelle sind von diesem Chip 8 Daten- und einige Steuerleitungen herangeführt.

Die Datenleitungen sind nicht nur als Ausgänge (wie für den Drucker), sondern auch als Eingänge verwendbar. Dabei kann für jedes der 8 Leitungen die Richtung einzeln bestimmt werden. Man kann somit z.B. drei Eingangs- und fünf Ausgangsleitungen definieren und somit einen externen Prozeß sowohl überwachen als auch steuern.

Bevor wir diese Verbindung zur Außenwelt des Amiga eingehend betrachten, muß erwähnt werden, daß die Anschlüsse des

Amiga-Parallelports mit TTL-Pegel arbeiten, d.h. es werden Spannungswerte von 0 Volt für LO und 5 Volt für HI ein- und ausgegeben. Wenn Sie also eine eigene Schaltung an den Port anschließen, sollten Sie diese Pegel nicht überschreiten. Außerdem kann mit den Ausgangssignalen keine große Leistung gesteuert werden, so daß Sie auf jeden Fall einen Treiber für Ihre Schaltung vorschalten sollten. Andernfalls könnte der PIA-Baustein zerstört werden!

Gehen wir also davon aus, daß Sie eine TTL-Schaltung an den Amiga angeschlossen haben, welche Ein- und Ausgänge besitzt. Nehmen wir hierfür als Beispiel eine ganz einfache Schaltung:

- ein Schalter, welcher Bit 0 des Ports auf LO-Pegel schalten kann, und
- eine Lampe bzw. Leuchtdiode, die über einen Treiber an Bit 1 des Ports angeschaltet werden kann.

Lassen Sie den Schalter zunächst in der Stellung stehen, in der er offen ist und somit keinerlei Pegel an den Port schaltet. Solange nämlich der Port auf Ausgang geschaltet ist, darf kein direkter Pegel dort angeschlossen werden, da sonst HI auf LO kommen könnte und so einen Kurzschluß im PIA entstünde. Die Lampe mit dem Treiber können Sie aber schon anschließen, sie wird dann leuchten.

Standardmäßig sind die 8 Datenleitungen auf Eingang geschaltet, da ja Signale anliegen könnten. Die Daten, die an diesem Port anliegen, finden Sie in dem Hardware-Register, welches das Datenregister des PIA darstellt. Sollten Sie jedoch bereits den Drucker benutzt haben, so ist die Richtung bereits auf Ausgang geschaltet. Ändern Sie dann den Inhalt der Speicherzelle $BFE301 mit dem 'm'-Befehl des SEKA auf 0 (Erklärung später).

Das Datenregister hat die Adresse $BFE101 und ist 8 Bit breit, also ein Byte. Diese Adresse muß auch mit einem Bytezugriff programmiert werden, da es sich ja hier um eine ungerade Adresse handelt.

Geben Sie nun einmal im SEKA 'm $bfe101' ein. Sie finden hier den Wert $FF, was bedeutet, daß alle Eingänge des Ports offen und damit auf logisch HI sind. Auch das Bit 1, an dem die Lampe hängt, ist auf 1, da TTL-Eingänge immer auf diesem Pegel sind.

Wenn Sie nun einen anderen Wert eingeben und Return drücken, so erwarten Sie wohl, daß sich der Inhalt der betreffenden Speicherzelle auch ändert. Wie Sie aber an der Anzeige erkennen, hat sich der Wert $FF nicht geändert, die Lampe ist auch nicht ausgegangen.

Der Grund hierfür ist einfach der, daß die Leitungen immer noch auf Eingang stehen und das Hardware-Register immer noch den Wert enthält, der den angeschlossenen Leitungspegeln entspricht. Drücken Sie daher erst einmal die Escape-Taste, um wieder in den Kommandomodus des SEKA zu gelangen.

Legen Sie nun mit dem Schalter ein LO-Signal auf Bit 0. Wenn Sie dann wieder mit 'q $bfe101' den Inhalt des Datenregisters betrachten, finden Sie hier tatsächlich den Wert $FE, was auf ein gelöschtes Bit 0 hinweist.

Um nun auch die Lampe an und ausschalten zu können, müssen wir das Bit 1 des Ports auf die Richtung 'Ausgang' schalten. Die Information, welche Bits auf Eingang und welche auf Ausgang geschaltet sind, ist im Datenrichtungs-Register des PIA enthalten. Dieses Register liegt an der Speicheradresse $BFE301. Dieses Register haben Sie eben noch auf Null gesetzt, um die Leitungen auf Eingang zu setzen.

Jedes Bit in diesem Byte entspricht einer der 8 Datenleitungen, Bit 0 für Leitung 0, Bit 1 für Leitung 1 usw. Wird das entsprechende Bit gelöscht, so wird die Leitung auf Eingang geschaltet, wird es gesetzt, haben wir eine Ausgangsleitung.

Dies wollen wir nun einmal an unserem Beispiel ausprobieren. Wir wollen alle Bits auf Eingang lassen, bis auf Bit 1, da an Leitung 1 ja die Lampe angeschlossen ist. Wir müssen also Bit 1 des Datenrichtungs-Registers auf 1 setzen. Dazu geben Sie bitte

'm $bfe301' ein, woraufhin der SEKA Ihnen den momentanen Inhalt dieses Registers angibt, eine Null. Geben Sie hier nun eine 2 ein und drücken Return. Wie Sie an der Anzeige sehen können, wird dieser Wert auch in das Register geschrieben.

Nun können Sie auch die Lampe ausschalten. Schreiben Sie dazu den Wert 0 in das Datenregister $BFE101: die Lampe geht aus.

Wenn Sie jetzt noch einmal dieses Datenregister auslesen, finden Sie dort den Wert $FD. Den Zustand des Bits 1 können Sie also hier wiederfinden, was manchmal recht praktisch ist, um festzustellen, was zuletzt in das Register geschrieben wurde.

Wir wollen nun ein kleines Programm schreiben, welches den Zustand des Schalters direkt auf die Lampe überträgt. Ist der Schalter ausgeschaltet, soll die Lampe leuchten, andernfalls nicht. Um das Programm zu beenden, drücken Sie den Feuerknopf eines an Port 2 angeschlossenen Joysticks. Das Programm dafür ist folgendes:

```
Datenreg  = $BFE101      ;Datenregister
DRichtreg = $BFE301      ;Datenrichtungs-Register
Feuer     = $BFE001      ;Feuerknopf in Bit 7

run:
     move.b  #2,DRichtreg     ;Bit 1 auf Ausgang
loop:
     move.b  Datenreg,d0      ;Eingangsdaten in D0
     lsl     d0               ;nach links schieben
     move.b  d0,Datenreg      ;und wieder ausgeben
     tst.b   Feuer            ;Test auf Feuerknopf
     bmi     loop             ;nicht gedrückt: weiter
     rts                      ;sonst Ende
```

Sie können nun den Schalter hin- und herschalten, die Lampe wird den Zustand anzeigen. Erreicht wurde dies durch den LSL-Befehl, welcher die Bits von D0 nach links schiebt. Dabei wird Bit 0 nach Bit 1 geschoben, was genau das ist, was wir wollten! Das Ergebnis in D0 kann dann nämlich direkt wieder in das Datenregister geschrieben werden und gibt in Bit 1 den Zustand des Schalters aus.

Eine kleine Aufgabe: Schreiben Sie dieses Programm einmal so um, daß die Lampe angeht, wenn der Schalter LO-Potential schaltet!

Für die Bedienung des Druckers reichen diese 8 Bit nicht aus. Es werden noch zusätzliche Signalleitungen benötigt, die anzeigen, daß ein gültiges Byte an den 8 Datenleitungen ansteht oder daß der Drucker bereit ist, Daten zu empfangen.

Diese Leitungen finden Sie im Register an der Adresse $BFD0FE. Die Bits 0 bis 2 sind direkt an dem Druckerport angeschlossen:

Bit 0 an BUSY
Bit 1 an POUT
Bit 2 an SEL des Druckerports

Sie können diese Leitungen also als zusätzliche Eingangs-Leitungen für Ihren Anschluß eines externen Gerätes verwenden.

5.5 Tonerzeugung

Ein wirklich interessantes Thema, gerade für den Maschinensprache-Programmierer, ist die Erzeugung von Tönen und Geräuschen. Der Amiga bietet hierfür die Möglichkeit, mit Hilfe des Audio.Devices und diverser I/O-Strukturen Töne, Geräusche und/oder Musikstücke im Hintergrund abspielen zu lassen. Diese Methode überlassen wir aber den C- oder BASIC-Programmierern, da wir mit einem kleinen Maschinenprogramm direkt die Audio-Hardware programmieren können!

Der im Amiga eingebaute Chip namens Paula nimmt uns dabei sehr viel Arbeit ab, da er alle Fähigkeiten zur Tonerzeugung besitzt. Da dieser Chip ebenfalls über Hardware-Register für den Prozessor erreichbar ist, können wir dies auch direkt ausnutzen. Schließlich kann keine Bibliothek irgendeiner Hochsprache mehr als wir tun: den Chip programmieren.

Wie funktioniert's? Nun, da der Chip über einen eigenen Speicherzugriff (DMA = direct memory access) verfügt, brauchen wir ihm nur zu sagen, wo wir die Daten des gewünschten Tones bzw. der Tonfolge im Speicher abgelegt haben. Außerdem müssen wir ihm mitteilen, wie er diese Daten zu interpretieren hat.

Beginnen wir mit dem einfachsten Fall: der Erzeugung eines konstanten Tones. Ein solcher Ton besteht ja aus einer Schwingung, welche sich ständig wiederholt. Tragen wir eine solche Schwingung in ein Diagramm ein, so sehen wir die Kurvenform dieser Schwingung. Es gibt dabei einige Standardkurven: Sinus, Rechteck, Dreieck oder Sägezahn. Die einfachste dabei ist die Rechteckschwingung.

Um einen Ton mit Rechteckkurve zu erzeugen, brauchen wir nur den Lautsprecher ein- und auszuschalten. Die Frequenz, in der dies geschieht, ist dann auch die Frequenz des hörbaren Tones.

Wir wollen nun dem Amiga einen solchen Ton entlocken. Dafür legen wir zuerst einmal eine Tabelle an, in der die Amplituden des zu erzeugenden Tones liegen. Für die Rechteckschwingung brauchen wir nur zwei Einträge in dieser Tabelle: einen großen und einen kleinen Wert. Da für den Sound-Chip im Amiga Amplitudenwerte von -128 bis +127 existieren, sieht unsere Tabelle also z.B. so aus:

```
soundtab:
  dc.b -100,100
```

Die Adresse dieser 'Tabelle' geben wir nun an den Sound-Chip weiter. Dafür haben wir 4 Möglichkeiten, da der Amiga 4 Sound-Kanäle besitzt. Die Adresse des Hardware-Registers, in die die Tabellenadresse für Kanal 0 hineingeschrieben werden muß, ist $DFF0A0, für Kanal 1 ist es $DFF0B0, für Kanal 2 $DFF0C0 und für Kanal 3 $DFF0D0. Bei Verwendung der Stereo-Ausgänge gilt: Kanal 0 und 3 steuern den linken, Kanal 1 und 2 den rechten Lautsprecher. Wir wählen für unser Beispiel den Kanal 0, und schreiben daher:

```
move.l #soundtab,$DFF0A0  ;Adresse der Tabelle
```

Als nächstes müssen wir dem Sound-Chip mitteilen, wie viele Daten in der Tabelle enthalten sind. Diese Daten werden immer wieder vom Anfang bis zum Ende gelesen und an den Lautsprecher weitergegeben. Ist er am Ende angelangt, fängt er wieder am Anfang an usw. Da der Sound-Chip sich diese Daten selbst aus dem Speicher holt und dies immer in Wortbreite geschieht, die Daten selbst aber Bytes sind, muß die Tabelle immer eine gerade Anzahl Bytes enthalten. Die Länge, die wir jetzt übermitteln werden, gibt dann auch die Anzahl der Worte, also Anzahl der Bytes/2, an.

Die Länge wird für Kanal 0 im Register an der Adresse $DFF0A4 übergeben (für Kanal x einfach x*$10 addieren!):

```
move    #1,$dff0a4        ;Länge der Tabelle in Worten
```

Nun folgt die Angabe, wie schnell die Daten ausgelesen und an den Lautsprecher ausgegeben werden sollen. Dieses Wort gibt somit die Frequenz an, allerdings in umgekehrtem Verhältnis: je höher der Wert, desto niedriger die Frequenz. Wählen wir hierfür einmal den Wert 600:

```
move    #600,$dff0a6      ;Abtastrate
```

Schließlich müssen wir noch die Lautstärke bestimmen, mit der der Ton oder das Geräusch ertönen soll. Dabei können wir zwischen 65 verschiedenen Lautstärken wählen. Nehmen wir als mittlere Lautstärke den Wert 40:

```
move    #40,$dff0a8       ;Lautstärke
```

So, das waren die Daten, die der Sound-Chip für den zu erzeugenden Ton braucht. Bis jetzt ist aber noch nichts passiert. Wie auch? Schließlich kann der Chip nicht wissen, ob die Daten, die in den Registern stehen, gültig sind und ob er überhaupt etwas von sich geben soll!

Werfen wir also die Tonausgabe bzw. die DMA an, damit wir auch etwas hören können. Dafür nehmen wir das DMA-Kontrollregister an der Adresse $DFF096. Wir brauchen von diesem Wort für unsere Zwecke nur 6 Bits:

Bit 15 ($8000) Ist dieses Bit gesetzt, so wird jedes Bit, welches hier eingeschrieben wird, im internen Register auch gesetzt, andernfalls werden diese Bits gelöscht. 0-Bits werden nicht beeinflußt. Dies ist sehr nützlich, da in diesem Wort auch DMA-Informationen für den Diskettenbetrieb enthalten sind, welche auch nicht verändert werden sollen.

Bit 9 ($200) Dieses Bit ermöglicht überhaupt erst den DMA-Speicherzugriff des Chips. Wenn wir den Ton starten wollen, müssen wir es also setzen.

Bit 0-3 Schalten Kanal 0-3 ein, wenn das Bit gesetzt ist.

Wir starten also unseren Ton, indem wir die Bits 15, 9 und 0 setzen:

```
        move    #$8000+$200+1,$dff096 ;DMA starten
```

Hier noch einmal die Tonerzeugung auf einen Blick, diesmal mit einem sinusähnlichen Ton:

```
;** Soundgenerierung über Hardware-Register **


ctlw  = $dff096              ;DMA-Control

c0thi = $dff0a0              ;Tabellenadresse HI
c0tlo = c0thi+2              ;Tabellenadresse LO
c0tl  = c0thi+4              ;Tabellenlänge
c0per = c0thi+6              ;Abtastrate
c0vol = c0thi+8              ;Lautstärke
```

```
run:                            ;* einfachen Ton erzeugen
     move.l  #table,c0thi       ;Tabellenanfang
     move    #8,c0tl            ;Tabellenlänge 8 Worte
     move    #400,c0per         ;Abtastrate
     move    #40,c0vol          ;Lautstärke

     move    #$8201,ctlw        ;DMA/Sound starten
     rts


table:                          ;Sound-Tabelle: Sinus
 dc.b -40,-70,-90,-100,-90,-70,-40,0
 dc.b 40,70,90,100,90,70,40,0
```

Um diesen Ton wieder auszuschalten, brauchen wir nur Bit 0 des DMA-Kontrollregisters zurückzusetzen. Dafür können wir, wie erwähnt, durch eine 0 in Bit 15 einstellen, daß alle anderen gesetzten Bits des in dieses Register geschriebenen Wortes im Chip selbst gelöscht werden. Um also Bit 0 zu löschen, schreiben wir dort lediglich eine 1 hinein: Bit 15=0 => Bit 0 löschen. Hier die kleine Routine, um den Ton aus Kanal 0 zu stoppen:

```
still:                          ;* Ton abschalten
     move    #1,ctlw            ;Kanal 1 abschalten
     rts
```

Hier noch eine kleine Anwendung dieser Routinen: ein Programm zur Erzeugung eines kurzen Pieptones, welchen man z.B. als Tastaturklick einsetzen kann:

```
;** Piepton-Erzeugung **


ctlw  = $dff096                 ;DMA-Control

c0thi = $dff0a0                 ;Tabellenadresse HI
c0tlo = c0thi+2                 ;Tabellenadresse LO
c0tl  = c0thi+4                 ;Tabellenlänge
c0per = c0thi+6                 ;Abtastrate
c0vol = c0thi+8                 ;Lautstärke
```

```
beep:                           ;* kurzen Piepton erzeugen
     move.l  #table,c0thi       ;Tabellenanfang
     move    #4,c0tl            ;Tabellenlänge
     move    #300,c0per         ;Abtastrate
     move    #40,c0vol          ;Lautstärke

     move    #$8201,ctlw        ;Start DMA bzw. Sound

     move.l  #20000,d0          ;Verzögerungszähler
loop:
     dbra d0,loop               ;herunterzählen

still:
     move #1,ctlw               ;und Ton abschalten
     rts

table:                          ;Sound-Tabelle
     dc.b -40,-70,-40,0,40,70,40,0
```

Sie können also auf diese Art leicht ein bis vier Töne gleichzeitig unabhängig voneinander erklingen lassen. Eine weitere Möglichkeit bietet der Amiga, diese Klänge interessanter klingen zu lassen: wir können die Töne modulieren.

Angenommen, Sie wollen einen Sirenenton erzeugen. Sie können dies natürlich dadurch bewirken, daß Sie die gesamte Heultonsequenz programmieren. Wie Sie sich aber wohl vorstellen können, wäre dies ein Riesenaufwand!

Einfacher wird es, wenn wir dafür zwei Tonkanäle verwenden. Nehmen wir als Beispiel Kanal 1 für den Grundton und Kanal 0 für dessen Modulation. Für die Sirene muß Kanal 0 dann die Hüllkurve des Heultons enthalten und mit der richtigen Geschwindigkeit das Auf- und Abschwellen des Tons vorgeben.

Sie haben dann drei Möglichkeiten, wie Sie Kanal 0 dem Kanal 1 überlagern. Sie können die Lautstärke von Kanal 0 steuern, dessen Abtastrate bzw. Frequenz oder auch beides. Für unser Beispiel brauchen wir also die Frequenzmodulation.

Ändern Sie jetzt obiges Programm folgendermaßen ab:

```
;** modulierte Soundgenerierung über Hardware-Register **


ctlw  = $dff096              ;DMA-Control
adkon = $dff09e              ;Audio-/Disk-Control

c0thi = $dff0a0              ;Tabellenadresse HI
c0tlo = c0thi+2              ;Tabellenadresse LO
c0tl  = c0thi+4              ;Tabellenlänge
c0per = c0thi+6              ;Abtastrate
c0vol = c0thi+8              ;Lautstärke


run:
     move.l  #table,c0thi+16  ;Tabellenanfang Kanal 1
     move    #8,c0tl+16       ;Tabellenlänge
     move    #300,c0per+16    ;Abtastrate
     move    #40,c0vol+16     ;Lautstärke

     move.l  #table2,c0thi    ;Tabellenanfang Kanal 0
     move    #8,c0tl          ;Tabellenlänge
     move    #60000,c0per     ;Abtastrate
     move    #30,c0vol        ;Lautstärke

     move    #$8010,adkon     ;Modulations-Modus: FM
     move    #$8203,ctlw      ;Start DMA
     rts

still:                        ;* Ton abschalten
     move    #$10,adkon       ;keine Modulation mehr
     move    #3,ctlw          ;Kanäle ausschalten
     rts

table:                        ;Daten für Grundton
 dc.b -40,-70,-90,-100,-90,-70,-40,0
 dc.b 40,70,90,100,90,70,40,0

table2:                       ;Daten für Modulation
 dc.w 400,430,470,500,530,500,470,430
```

Wenn Sie dieses Programm starten, hören Sie einen Sirenenton, den Sie natürlich noch beliebig verändern können.

Wie Sie sehen, ist hierbei ein neues Register aufgetaucht: 'adkon'. Dieses Register kontrolliert sowohl die Modulation der

Audio-Kanäle als auch die Diskettenfunktion. Da hierbei jedoch die gleiche Technik wie beim DMA-Kontrollregister vorliegt, in dem nur die spezifizierten Bits gesetzt bzw. gelöscht werden (auch hier abhängig von Bit 15), brauchen wir uns um die Disketten-Bits nicht zu kümmern. Ich rate aber, damit nicht zu experimentieren!

Außer dem Kontrollbit 15 sind für unsere Zwecke noch die Bits 0-7 interessant. In diesen Bits wird nämlich bestimmt, welcher Audio-Kanal welchen anderen wie moduliert. Dabei gibt es allerdings eine Einschränkung:

Ein Kanal kann nur den nächsthöheren Kanal modulieren. Aus diesem Grund ist in obigem Beispiel auch Kanal 1 für den Grundton und Kanal 0 für die Modulation gewählt. Andersherum ist es ebensowenig möglich, wie z.B. mit Kanal 0 den Kanal 3 zu modulieren. Daher ist Kanal 3 auch nicht zur Modulation eines anderen verwendbar.

Hier eine Übersicht über die Bits 0-7 des 'adkon'-Registers mit deren Funktionen:

Bit	**Funktion**
0	Kanal 0 moduliert die Lautstärke von Kanal 1
1	Kanal 1 moduliert die Lautstärke von Kanal 2
2	Kanal 2 moduliert die Lautstärke von Kanal 3
3	Kanal 3 ausschalten
4	Kanal 0 moduliert die Frequenz von Kanal 1
5	Kanal 1 moduliert die Frequenz von Kanal 2
6	Kanal 2 moduliert die Frequenz von Kanal 3
7	Kanal 3 ausschalten

In unserem Beispiel haben wir Bit 4 gesetzt und somit Kanal 0 zur Frequenzmodulation von Kanal 1 bestimmt.

Wenn wir einen Kanal als Modulationskanal bestimmen, ändern sich einige Bedeutungen der Parameter dieses Kanals. So ist die Angabe der Lautstärke dieses Kanals unnötig und kann somit

entfallen. Außerdem sind die Daten der Tabelle nicht mehr als Bytes, sondern als Worte anzusehen. Diese Worte werden dann mit der bestimmten Rate in das vorgegebene Register des modulierten Kanals (hier: Abtastraten-Register) eingetragen.

Wenn Sie sowohl die Frequenz als auch die Lautstärke des anderen Kanals modulieren wollen (hier: Bits 0 und 4 von 'adkon' setzen), werden die Daten noch ein wenig anders interpretiert. Das erste Wort der Tabelle wird dann als Lautstärke, das zweite Wort als Abtastrate interpretiert und so weiter, also immer abwechselnd. Damit können Sie z.B. den Sirenenton eines vorbeifahrenden Polizeiwagens erzeugen.

5.6 Übersicht der Hardware-Register

Die folgenden Tabellen sollen Ihnen einen Überblick über die wichtigsten Hardware-Register geben. Die Beschreibung jedes einzelnen Registers würde hier den Rahmen sprengen, so daß ich hierfür auf entsprechende Literatur verweisen muß. Sollten Sie mit diesen Registern experimentieren wollen, so bedenken Sie, daß dies evtl. zu einem Absturz des Rechners führen kann. Sichern Sie daher vorher immer Ihre Daten auf der Diskette und nehmen Sie diese dann vorsichtshalber aus dem Laufwerk, da ja auch dieses vielleicht Fehlfunktionen ausführen könnte.

Beginnen wir mit den bereits teilweise besprochenen PIAs. Es handelt sich dabei um den PIA-Typ 8520, über welchen auch Datenblätter existieren. Solche Datenblätter kann ich nur empfehlen, da sie wirklich alle Möglichkeiten des Bausteins aufzeigen. Sie müssen nur beachten, daß einige Funktionen und Anschlüsse der 8520 bereits fest im Amiga integriert sind und somit für die Benutzung der PIAs Einschränkungen auferlegen.

PIA A	PIA B	Register-Bedeutung
BFE001	*BFE000*	Datenregister A
BFE101	*BFE100*	Datenregister B
BFE201	*BFE200*	Datenrichtungsregister A
BFE301	*BFE300*	Datenrichtungsregister B
BFE401	*BFE400*	Taktgeber A LO
BFE501	*BFE500*	Taktgeber A HI
BFE601	*BFE600*	Taktgeber B LO
BFE701	*BFE700*	Taktgeber B HI
BFE801	*BFE800*	Ereignisregister Bits 0-7
BFE901	*BFE900*	Ereignisregister Bits 8-15
BFEA01	*BFEA00*	Ereignisregister Bits 16-23
BFEB01	*BFEB00*	unbenutzt
BFEC01	*BFEC00*	serielles Datenregister
BFED01	*BFED00*	Interrupt-Kontrollregister
BFEE01	*BFEE00*	Kontrollregister A
BFEF01	*BFEF00*	Kontrollregister B

Einige interne Bedeutungen:

$BFE101 Datenregister für die Parallel-Schnittstelle

$BFE301 Datenrichtungsregister für die Parallel- Schnittstelle

$BFEC01 Zustand der Tastatur, enthält die zuletzt gedrückte Sondertaste (Shift, Alternate, Control, Amiga)

Nun folgen die Register, welche für die Tonerzeugung zuständig sind. Hierbei sind besonders die ersten beiden Register mit Vorsicht zu behandeln, da sie bei Fehlbehandlung unschöne Effekte auslösen können.

Diese Register können entweder nur gelesen oder nur beschrieben werden. Dies wird durch die R/W-Angabe vorgegeben, welche in der Liste steht.

Adresse	R/W	Bedeutung
DFF096	*W*	DMA-Control schreiben
DFF002	*R*	DMA-Control und Blitter-Statuslesen
-- Audiokanal 0 --		
DFF0AA	*W*	Datenregister
DFF0A0	*W*	Zeiger auf Tabellenanfang Bits 16-18
DFF0A2	*W*	Zeiger auf Tabellenbeginn Bits 0-15
DFF0A4	*W*	Tabellenlänge
DFF0A6	*W*	Abtastrate / Periode
DFF0A8	*W*	Lautstärke
-- Audiokanal 1 --		
DFF0BA	*W*	Datenregister
DFF0B0	*W*	Zeiger auf Tabellenanfang Bits 16-18
DFF0B2	*W*	Zeiger auf Tabellenbeginn Bits 0-15
DFF0B4	*W*	Tabellenlänge
DFF0B6	*W*	Abtastrate / Periode
DFF0B8	*W*	Lautstärke
-- Audiokanal 2 --		
DFF0CA	*W*	Datenregister
DFF0C0	*W*	Zeiger auf Tabellenanfang Bits 16-18
DFF0C2	*W*	Zeiger auf Tabellenbeginn Bits 0-15
DFF0C4	*W*	Tabellenlänge
DFF0C6	*W*	Abtastrate / Periode
DFF0C8	*W*	Lautstärke
-- Audiokanal 3 --		
DFF0DA	*W*	Datenregister
DFF0D0	*W*	Zeiger auf Tabellenanfang Bits 16-18
DFF0D2	*W*	Zeiger auf Tabellenbeginn Bits 0-15
DFF0D4	*W*	Tabellenlänge
DFF0D6	*W*	Abtastrate / Periode
DFF0D8	*W*	Lautstärke

Nun zu den Registern, in denen die Informationen über Joystick, Maus oder Potentiometer enthalten sind. Diese Adressen sind teilweise bereits aufgetaucht.

Adresse	**R/W**	**Bedeutung**
DFF00A	*R*	Joystick/Maus Port 1
DFF00C	*R*	Joystick/Maus Port 2
DFF012	*R*	Potentiometerpaar 1 Zähler
DFF014	*R*	Potentiometerpaar 2 Zähler
DFF018	*R*	Potentiometer-Anschlüsse Eingang
DFF034	*W*	Potentiometer-Port Richtung

6. Arbeiten mit dem Amiga-Betriebssystem

Jetzt wollen wir einen weiteren Schritt zur sinnvollen Assembler-Programmierung tun, indem wir den Amiga in unsere Wünsche einweihen. Es genügt schließlich nicht, einen Text irgendwo im Speicher liegen zu haben, man will ihn ja auch auf dem Bildschirm ausgeben können. Aber wissen Sie, wie man einen Buchstaben ins Bild bringt? Oder gar, wie man ein Fenster auf dem Bildschirm zeichnet, das sich sogar von der Maus verändern läßt?

Nun, diese Dinge brauchen wir auch gar nicht so genau zu wissen. Glücklicherweise verfügt das Betriebssystem des Amiga bereits über Routinen, die uns solche Kleinarbeit abnehmen. Diese Routinen sind zu sogenannten Bibliotheken zusammengefaßt, die wir uns einmal genauer ansehen sollten.

6.1 Bibliotheken laden

Bevor wir eine solche Bibliothek anwenden können, muß sie erst einmal verfügbar gemacht, d.h. in den Speicher geladen werden. Dabei wird leider immer die ganze Bibliothek geladen, auch wenn wir nur eine der Funktionen benötigen.

Zunächst einmal muß man sich überlegen, was das eigene Programm alles können muß, also welche Bibliotheken man überhaupt braucht. Für die einfache Ein- und Ausgabe von Texten brauchen wir ja keine Bibliothek, welche Routinen für bewegte Graphik enthält!

Es gibt auf einer normalen Workbench-Disk eine ganze Anzahl Bibliotheken. Hier eine Übersicht über deren Namen und Anwendungsgebiete:

exec.library

Diese Bibliothek raucht man schon zum Laden der anderen Bibliotheken. Sie ist daher bereits im Speicher und braucht somit nicht geladen werden. Mit ihr werden grundlegende Funktionen wie Speicher reservieren oder I/O- Kanäle bearbeiten möglich.

dos.library

Enthält alle Funktionen für normale Ein- /Ausgabe-Operationen wie Bildschirm- oder Disketten-Bedienung.

intuition.library

Dient zur Bearbeitung von Screens, Fenstern, Menüs usw.

clist.library

Bietet Routinen zur Bearbeitung von Copper- Lists, mit denen der Bildaufbau gesteuert werden kann.

console.library

Enthält Graphik-Routinen für die Textausgabe in Consolen-Fenstern.

diskfont.library

Wird für die Verwendung von Zeichensätzen, die auf der Diskette gespeichert sind, benötigt.

graphics.library

Enthält Funktionen zur Steuerung des Blitters bzw. des Graphik-Chips und wird daher für die graphischen Grundfunktionen benötigt.

icon.library

Wird für die Erstellung und Bearbeitung von Workbench-Symbolen verwendet.

layers.library

Ist vorgesehen für die Bearbeitung des Bildschirmspeichers (Layers).

mathffp.library

Enthält grundlegende mathematische Fließkomma-Operationen.

mathieeedoubbas.library

Bietet mathematische Grundfunktionen für ganze Zahlen (Integers).

mathtrans.library

Enthält höhere Funktionen wie Winkel- oder Logarithmus-Berechnung.

potgo.library

Dient für die Auswertung von analogen Eingaben in den Amiga.

timer.library

Stellt Routinen für zeitkritische Programme zur Verfügung, mit denen exakte Zeitintervalle programmiert werden können.

translator.library

Enthält als einzige Funktion 'Translate', die einen normalen Text in die phonetische Schreibweise für den Narrator, den Sprachsynthesizer, umsetzt.

Man könnte natürlich immer alle diese Bibliotheken öffnen bzw. einladen. Allerdings muß man bedenken, daß dies erstens Zeit und zweitens Speicherplatz kostet. Aus diesem Grund sollte man sich immer gut überlegen, welche Funktionen man benötigt und in welchen Bibliotheken sich diese befinden.

Nehmen wir nun als Beispiel einmal an, wir wollen ein Programm schreiben, welches Textein-/ausgabe ausführen soll. Dafür benötigen wir also die Bibliothek 'dos.library', welche wir nun laden wollen.

Dieser Ladevorgang ist eine Aufgabe der 'exec.library'. Diese Bibliothek enthält u.a. die Funktion 'OpenLib', welche mit Angabe der benötigten Parameter aufgerufen werden muß.

6.2 Funktionen aufrufen

Jede Library beginnt im Speicher mit einer Reihe JMP-Befehle. Diese JMP's liegen schön der Reihe nach und verzweigen in die Routinen, die die Bibliothek enthält. Um also eine Funktion aufzurufen, muß man zunächst den Anfang dieser JMP-Tabelle finden, um von dort aus die Funktion x mit einem Sprung zum x-ten JMP aufzurufen. Üblicherweise wird dies mit einem Offset bewerkstelligt, mit dem ein JSR die Funktion startet. Da man normalerweise nicht den Anfang der JMP-Tabelle, sondern dessen Ende als Basisadresse erhält, sind diese Offsets negativ.

In der Praxis sieht dies recht einfach aus. Um nun nach obigen Beispiel die 'dos.library' zu laden, brauchen wir also zunächst die Basisadresse der 'exec.library'. Diese ist die Adresse $000004. Beim Aufruf einer Funktion aus einer anderen Bibliothek muß natürlich eine andere Basisadresse verwendet werden, welche man auch beim Laden der Library erhält.

Nun brauchen wir den Offset für die gewünschte Funktion. Es handelt sich wie gesagt um die Funktion OpenLib, welche den Offset -408 trägt. Eine Übersicht aller Funktions-Offsets finden Sie übrigens im Anhang.

Für die OpenLib-Funktion brauchen wir jetzt nur noch einen Zeiger auf den Namen der zu ladenden Library, hier 'dos.library', und ein Speicher-Langwort, in das wir die Basisadresse der DOS-Bibliothek, welche die OpenLib-Funktion zurückgibt, ablegen können. Wichtig ist übrigens, daß der Bibliotheksname wirklich in Kleinbuchstaben angegeben wird, da das Öffnen sonst nicht funktioniert. Ich hatte zuerst diese Namen in Großbuchstaben angegeben und lange den 'Fehler' meines Programms gesucht.

Und so sieht all dies als Programm aus:

```
;** Laden der DOS-Bibliothek 'dos.library' **

execbase = 4          ;Basisadresse der EXEC-Bibliothek
OpenLib  = -408       ;Offset der OpenLib-Funktion
```

```
start:
     move.l    execbase,a6     ;Basisadresse in A6
     lea       dosname,a1      ;Adresse vom Bibliotheks-Namen
     moveq     #0,d0           ;Versionsnummer
     jsr       OpenLib(a6)     ;Aufruf der Funktion
     move.l    d0,dosbase      ;DOS-Basisadresse retten
     beq       error           ;wenn 0, dann Fehler !

     ...
     ...                       ;weiteres Programm...

error:
     ...                       ;Fehler aufgetreten
     ...

dosname:                       ;Name der zu öffnenden Bibliothek
     dc.b      'dos.library',0,0 even
dosbase:                       ;Platz für DOS-Basisadresse
     blk.l     1
```

Auf diese Art und Weise wird die DOS-Bibliothek geladen und verfügbar gemacht. Alle anderen Bibliotheksfunktionen werden auf diese Weise aufgerufen, wobei die Parameter in verschiedenen Registern hin und her übergeben werden. Auch ist meistens das Datenregister D0 im Fehlerfall Null, also wenn die Funktion nicht ordnungsgemäß ausgeführt wurde.

Wenn Ihr Programm nun seine Arbeit getan hat, so muß vor der Rückkehr in den CLI oder die Workbench zuerst einmal jede geöffnete Bibliothek wieder geschlossen werden. Dies übernimmt die Funktion CloseLib mit dem Offset -414. Sie liegt ebenso wie OpenLib in der EXEC-Bibliothek und benötigt als Parameter nur die Basisadresse der zu schließenden Bibliothek. So sieht dann nach obigem Öffnen der Programmteil zum Schließen der 'dos.library' so aus:

```
CloseLib = -414

     ...
     move.l    execbase,a6     ;EXEC-Basisadresse
     move.l    dosbase,a1      ;DOS-Basisadresse
     jsr       CloseLib(a6)    ;Bibliothek schließen
```

6.3 Programmvorbereitung

Bevor man ein Programm an die Arbeit gehen lassen kann, muß man üblicherweise erst einmal einige Vorbereitungen treffen. Damit ist nicht das Kochen der obligatorischen Tasse Kaffee gemeint, sondern Initialisierungen, die das Programm ausführen muß.

Nehmen wir einmal als Beispiel ein Programm, welches eine kleine Textverarbeitung darstellen soll. Ein solches Programm muß Texte speichern können, wozu natürlich ein Speicherbereich benötigt wird. Außerdem sollen Tastatureingaben und Bildschirmausgaben stattfinden, wofür ein geeignetes Ausgabefenster benötigt wird.

Für diese Anforderungen müssen zunächst einmal eine oder mehrere Bibliotheken geöffnet werden, was wir bereits besprochen haben. Nehmen wir an, die DOS-Bibliothek sei geladen, damit wir die nächsten Schritte in Angriff nehmen können.

6.3.1 Speicher reservieren

Um vom Betriebssystem einen Speicherbereich zugeordnet zu bekommen, gibt es mehrere Möglichkeiten bzw. Funktionen. Eine davon müssen wir verwenden, damit es beim gleichzeitigen Ablauf mehrerer Programme (Multitasking) nicht zu Überschneidungen der verwendeten Speicherbereiche kommen kann.

Betrachten wir zuerst die Funktion, welche im Normalfall angewendet wird. Diese Funktion liegt in der residenten EXEC-Bibliothek und trägt den Namen AllocMem (Offset -$c6). Sie reserviert einen Speicherbereich der gewünschten Länge (in D0), welcher dann irgendwo im Speicher liegt. Die wirkliche Adresse, ab welcher der Speicher reserviert wird, erhält man im Datenregister D0 zurück. Ist dieser Rückgabewert 0, so konnte die gewünschte Speicherlänge nicht reserviert werden.

Zusätzlich zur Speicherlänge wird an die Funktion noch ein Moduswort in D1 übergeben. In diesem Moduswort kann bestimmt werden, ob der reservierte Speicher direkt gelöscht wird oder nicht.

Und so sieht das ganze als Programmteil aus:

```
execbase = 4
AllocMem = -$c6

    ...
    move.l    #anzahl,d0      ;zu reservierende Anzahl Bytes
    move      #modus,d1       ;Moduswort
    move.l    execbase,a6     ;DOS-Basisadresse in A6
    jsr       AllocMem(a6)    ;Funktions-Aufruf
    move.l    d0,adresse      ;Speicheranfang retten
    beq       fehler          ;Speicher nicht reserviert!
    ...
```

Die zweite Möglichkeit der Speicher-Reservierung ist eine Funktion namens AllocAbs (Offset -$CC). Hiermit wird im Gegensatz zur AllocMem-Funktion ein ganz bestimmter Speicherbereich reserviert. Dazu wird in D0 die Anzahl zu reservierenden Bytes und im Adreßregister A1 die gewünschte Startadresse übergeben. Auch diese Funktion gibt in D0 eine 0 zurück, wenn dieser Speicherbereich nicht reserviert werden konnte.

```
execbase = 4
AllocAbs = -$cc

    ...
    move.l    #anzahl,d0      ;zu reservierende Anzahl Bytes
    lea       adresse,a1      ;gewünschte Startadresse
    move.l    execbase,a6     ;EXEC-Basisadresse
    jsr       AllocAbs(a6)    ;Speicher reservieren
    tst.l     d0              ;alles OK ?
    beq       fehler          ;nein !
    ...
```

Wenn das Programm seine Arbeit getan hat, muß vor der Rückkehr ins CLI bzw. Workbench der reservierte Speicher wieder ans System zurückgegeben werden. Dazu dient die Funktion FreeMem (Offset -$D2).

Dieser Funktion wird ebenso wie AllocAbs die Byte-Anzahl in D0 sowie die Startadresse des Speichers in A1 übergeben. Dabei ist allerdings eines besonders zu beachten:

Das Abmelden eines Speicherbereiches, welcher nicht reserviert war, führt normalerweise zum Absturz des Rechners (der Guru meditiert...)!

Und so sieht die Befreiung des gefangenen Speichers im Programm aus:

```
execbase = 4
FreeMem  = -$d2

    ...
    move.l    #anzahl,d0      ;abzumeldende Anzahl Bytes
    lea       adresse,a1      ;Startadresse von AllocAbs
    move.l    execbase,a6     ;EXEC-Basisadresse
    jsr       FreeMem(a6)     ;Speicher freigeben
    tst.l     d0              ;alles OK ?
    beq       fehler          ;nein ! Nanu ?
    ...
```

6.3.2 Öffnen eines Fensters, einfache Art

Sie werden sich vielleicht über die eigenartige Überschrift dieses Kapitels wundern. Der Unterschied zwischen den zwei grundlegenden Arten, ein Fenster zu öffnen, ist jedoch so groß, daß nicht beide in einem Kapitel behandelt werden sollten.

Die hier vorgestellte Art, ein Fenster zu öffnen, ist zwar sehr einfach, bietet jedoch nicht die Möglichkeit, mit beliebigen Gadgets zu arbeiten. Diese Gadgets sind z.B. das Schließsymbol in der linken oberen Ecke mancher Fenster oder das Größenänderungssymbol rechts unten.

Öffnet man auf die einfache Art ein Fenster, so sind fast alle Gadgets da, das Schließsymbol fehlt allerdings. Aus diesem Grund eignet sich diese Methode nicht für jede Anwendung. Doch nun zu der Methode selbst:

Wir verwenden zum Öffnen eines Fensters eine Funktion der DOS-Bibliothek, die zuvor natürlich geöffnet werden muß (s. Kapitel 'Bibliotheken laden'). Diese Funktion ist die allgemeine OPEN-Funktion, mit der man eine Menge anfangen kann. Aus diesem Grund empfiehlt es sich, in einem Programm eine 'open'-Unterroutine einzubauen, welche man dann öfters mal braucht. Zunächst einmal die grundlegenden Schritte bis hierher:

```
;** Laden der DOS-Bibliothek 'dos.library' **

execbase = 4          ;Basisadresse der EXEC-Bibliothek
OpenLib  = -408       ;Offset der OpenLib-Funktion
Open     = -30        ;Offset der DOS-Funktion OPEN

start:
     move.l    execbase,a6    ;Basisadresse in A6
     lea       dosname,a1     ;Adresse vom Bibliotheks-Namen
     moveq     #0,d0          ;Versionsnummer: egal
     jsr       OpenLib(a6)    ;Aufruf der Funktion
     move.l    d0,dosbase     ;DOS-Basisadresse retten
     beq       error          ;wenn 0, dann Fehler !

     ...
     ...                      ;nun Fenster öffnen usw.
     ...

error:
     ...                      ;Fehler aufgetreten
     ...

do_open:                      ;allgemeine OPEN-Funktion
     move.l    dosbase,a6     ;DOS-Basisadresse in A6
     jsr       Open(a6)       ;OPEN-Funktion aufrufen
     tst.l     d0             ;Test auf OK
     rts                      ;fertig, Test später auswerten
```

```
dosname:                        ;Name der zu öffnenden Bibliothek
    dc.b      'dos.library',0,0 even
dosbase:                        ;Platz für DOS-Basisadresse
    ds.l      1
```

Die Routine trägt hier den Namen 'do_open', da der Label 'Open' bereits für den Offset verwendet wurde. Diese Routine ruft lediglich die Open-Funktion auf, die in der DOS-Bibliothek steckt.

Natürlich ist dies noch nicht alles. Der Funktion müssen nun noch Parameter übergeben werden, die angeben, was denn überhaupt geöffnet werden soll. Diese Parameter werden in den Registern D1 und D2 übergeben, und zwar folgende:

D1 zeigt auf einen Definitionsblock dessen, was geöffnet werden soll. Dies bedeutet nichts anderes, als daß dort ein Dateiname stehen muß, der mit einem Null-Byte abgeschlossen ist. D1 muß natürlich wie alle Adressen als Langwort übergeben werden.

D2 enthält den Modus, mit dem diese Funktion ausgeführt werden soll. Hierbei wird unterschieden zwischen Modus Alt (1005) und Neu (1006). Diese Zahlen müssen dann jeweils im Langwort D2 übergeben werden.

Zurück zu der Absicht, ein Fenster zu öffnen. Glücklicherweise bietet das Amiga-DOS die Möglichkeit, Ein- und Ausgabekanäle auf ein- und dieselbe Art und Weise zu verwenden. Zu diesen Standardkanälen gehören außer Diskettendateien noch die Konsole (Tastatur und Bildschirm), die Druckerschnittstelle und die serielle RS232-Schnittstelle.

Interessant ist nun für uns die Konsolenein-/ausgabe. Wenn wir nämlich als Dateiname des zu öffnenden Kanals die Konsole angeben, so wird automatisch ein Fenster geöffnet.

Diese Angabe wird damit gemacht, daß man den Namen mit CON: beginnt, ähnlich zu DF0: bei Diskettenoperationen. Zusätzlich werden allerdings noch einige andere Angaben über das zu öffnende Fenster benötigt.

Diese Angaben sind X- und Y-Position der linken oberen Ecke des Fensters sowie der rechten unteren Ecke und der Name, der in der Titelzeile des Fensters stehen soll. Ein vollständiger Definitionsblock für ein solches Fenster sieht somit folgendermaßen aus:

```
fensternamе:  dc.b 'CON:0/10/640/200/** Test-Fenster **',0
```

Um dieses Fenster zu öffnen, muß also diese Zeile sowie folgende Programmzeilen in obiges Programm eingefügt werden:

```
        ...
        move.l    #fensternamе,d1     ;Zeiger auf Definition
        move.l    #1005,d2            ;Modus: alt
        bsr       do_open             ;Fenster öffnen
        beq       error               ;hat nicht geklappt!!
        move.l    d0,handle           ;sonst Kennummer retten
        ...

handle:         blk.l 1               ;Platz für Kennummer
```

Hier sind zwei Punkte noch zu klären:

Der übergebene Modus soll beim Öffnen eines solchen Fensters mit Alt angegeben werden. Da das Fenster vor dem Öffnen logischerweise noch nicht existiert, ist dies ziemlich unverständlich, aber stört ja nicht...

Der aus der 'do_open'-Routine zurückgegebene Parameter im Register D0 ist im Fehlerfall, also wenn das Öffnen nicht geklappt hat, Null. Andernfalls stellt dieser Wert die Identifikationsnummer des geöffneten Kanals (hier des Fensters) dar, die sogenannte Handle-Nummer, kurz Handle genannt. Diese Nummer muß gut verwahrt werden, da sie für jede Funktion, die auf diesen Kanal zugreifen soll, immer die Handle-Nummer mit angegeben werden muß! Im Beispiel wird die Nummer in dem Langwort 'handle' gespeichert.

Das so geöffnete Fenster besitzt, wie erwähnt, zwar kein Schließungssymbol, aber es läßt sich beliebig vergrößern bzw. verkleinern, verschieben oder nach vorne bzw. hinten hinter an-

dere Fenster legen. Diese Manipulationen, die man mit der Maus durchführen kann, werden vollständig vom Amiga bearbeitet (im Gegensatz zum ATARI ST, bei dem ein Programm sich auch teilweise um diese Dinge kümmern muß).

Eine wichtige Funktion, die diese Nummer benötigt, ist diejenige, welche den Kanal und damit in unserem Fall das Fenster wieder schließt. Diese Funktion liegt ebenso wie die Open-Funktion in der DOS-Bibliothek und trägt bezeichnenderweise den Namen 'Close'. Sie besitzt den Offset -36 und benötigt nur einen Parameter: die Handle-Nummer des zu schließenden Kanals im Register D1.

Um also nach getaner Arbeit sein Fenster wieder zu schließen, benötigt man folgende Zeilen in seinem Programm:

```
Close = -36

    ...
    move.l    handle,d1       ;Handle-Nummer in D1
    move.l    dosbase,a6      ;DOS-Basisadresse in A6
    jsr       Close(a6)       ;und Kanal schließen!
```

Und weg ist unser Fenster!

Noch eine Bemerkung zum Öffnen und Schließen von Fenstern auf diese Art: Öffnet man mehrere Fenster auf die gleiche Art und Weise, so erhält man auch mehrere Fenster und damit mehrere Handle-Nummern. Somit kann man beliebig viele Fenster auf dem Bildschirm darstellen und bearbeiten, selbstverständlich auch beliebige Fenster wieder schließen.

Nun gibt es noch eine weitere Art, ein Fenster einfach zu öffnen. Man übergibt dabei lediglich als Kanalbezeichnung anstelle von CON: den Namen RAW:. Alle anderen Parameter und Operationen bleiben gleich.

Wenn Sie dies nun einmal ausprobieren, werden Sie keinen Unterschied zwischen den beiden Arten von Fenstern feststellen. Beide Fenster sehen auch identisch aus und lassen sich auf gleiche Art mit der Maus bearbeiten. Der Unterschied tritt erst

dann zutage, wenn man in diesem Fenster Eingaben macht. Dann fällt nämlich auf, daß die Cursortasten wirklich funktionieren, im Gegensatz zu CON:-Fenstern wie das des CLI, in denen diese Tasten einfach ignoriert werden.

6.4 Ein- und Ausgabe

Neben der Verwaltung und Verrechnung von Daten ist wohl die wichtigste Arbeit eines Programms, diese Daten ein- und auszulesen. Dazu gehören alle Arten von Datenübertragung in und aus dem Speicher, wie z.B. Bildschirm- oder Druckerausgaben, Tastatureingaben, die Bedienung der seriellen oder der parallelen Schnittstelle, Ton- bzw. Sprachausgaben und nicht zuletzt die Diskettenoperationen.

Wir wollen nun nach und nach all diese Methoden der Datenein- und -ausgabe in der Programmierung und Anwendung kennenlernen und ausprobieren. Dabei werden einige Programme entstehen, die durch ihren Aufbau als Unterroutinen auch für spätere Programme nützlich sind. Es empfiehlt sich daher, eine Bibliothek mit solchen Unterroutinen anzulegen, die dann entweder direkt in ein neues Programm integriert oder dazugelinkt werden können.

Die Voraussetzung für irgendwelche Ein-/Ausgaben ist natürlich, daß man erst einmal Daten zum Ausgeben oder Platz für Eingabedaten vorbereitet hat. Um dies vorzubereiten, muß zunächst einmal ein korrekter Programmbeginn programmiert werden, in dem die EXEC- und die DOS-Bibliothek geöffnet werden und Speicherplatz reserviert wird. Ist dies geschehen, so beginnen die meisten Programme damit, einen Text auszugeben. Dieser Text kann eine Programm-Überschrift oder auch eine Aufforderung zur Eingabe von Daten über die Tastatur, ein sogenannter Prompt, sein. Beginnen wir daher ebenfalls mit der Technik der Bildschirmausgaben.

6.4.1 Bildschirmausgaben

Bei einem Rechner wie dem Amiga taucht die erste Frage schon hier auf: wohin soll die Bildschirmausgabe gehen? Bei anderen Rechnern ist dies trivial, existiert doch nur ein Bildschirm, auf den die Ausgaben stattfinden können. Beim Amiga dagegen muß man schon genau bestimmen, in welches Fenster geschrieben werden soll.

Dafür gibt es nun zwei Möglichkeiten:

1. Ausgaben in das CLI-Fenster
2. Ausgaben in ein eigenes Fenster

Die erste Möglichkeit hängt natürlich davon ab, ob man das Programm, welches die Ausgaben machen soll, überhaupt vom CLI aus gestartet hat. Wenn nicht, so müssen wir ohnehin ein eigenes Fenster für unser Programm öffnen. Wenn doch, so können wir das Fenster, welches der CLI bereits geöffnet hat, auch für unsere Ausgaben verwenden.

Für die zweite Möglichkeit ist also zuerst einmal ein Fenster zu öffnen. Wie wir bereits gesehen haben, gibt es dafür drei Methoden. Für die einfache Ausgabe von Zeichen und Texten ist der Unterschied zwischen diesen drei Fensterarten nicht so bedeutend, sodaß wir hier noch freie Hand haben, welche Art Fenster wir nun verwenden. Ich schlage daher vor, daß wir ein CON:-Fenster öffnen und dessen Handle-Nummer in 'conhandle' ablegen.

Wir haben also unser Fenster geöffnet und wollen eine Überschrift ausgeben. Wir überlegen uns dazu einen Text, welcher ausgegeben werden soll, und schreiben in das Programm eine entsprechende Zeile, mit der der Text mit seinem Null-Abschluß in den Speicher gebracht wird. Eine solche Zeile sieht dann etwa so aus:

```
titel: dc.b "** Willkommen zu diesem Programm ! **"
titelende:
even
```

Das 'even' ist eine kleine Zugabe, welche jedoch durchaus sinnvoll ist, wenn nach diesem Text entweder Wort-Daten oder gar Programmzeilen folgen. Die 'even'-Anweisung veranlasst den Assembler, hier bei Bedarf ein weiteres Nullbyte einzufügen, damit die nächste Adresse wieder gerade ist.

Um diesen Text nun auszugeben, brauchen wir eine weitere DOS-Funktion: Write. Diese hat den Offset -48 und benötigt drei Parameter:

In D1 die Handle des geöffneten Ausgabekanals, in den geschrieben werden soll. In unserem Fall ist dies die vom Open-Befehl erhaltene Handle-Nummer des Fensters.

In D2 die Adresse des auszugebenden Textes, in unserem Beispiel die Adresse 'titel', und

In D3 die Anzahl der auszugebenden Zeichen in Bytes.

Um die Anzahl der auszugebenden Bytes zu erhalten, müßten wir die Zeichen in unserem Text zählen. Um dies zu umgehen, ist der Label 'titelende' dazugefügt worden. Mit diesem Label kann der Assembler die Länge unseres Textes selber ausrechnen (wozu hat man denn einen Rechner?), wenn man schreibt:

```
move.l    #titelende-titel,d3
```

Die Notwendigkeit der Angabe der Textlänge hat übrigens den Vorteil, daß zwischen Anfang und Ende des Textes auch Steuerzeichen liegen können, mit denen direkt einige Dinge mit der Textausgabe selbst auf einen Schlag erledigt werden können. Diese Steuerzeichen werden wir später noch kennenlernen.

Hier nun den gesamten Programmteil:

```
Write = -48

    ...                         ;Fenster öffnen
    ...
    move.l  dosbase,a6          ;DOS-Basisadresse
    move.l  conhandle,d1        ;Handle übergeben
```

```
        move.l  #titel,d2           ;Textadresse
        move.l  #titelende-titel,d3 ;und Länge
        jsr     Write(a6)           ;Funktion aufrufen
        ...

titel:  dc.b  "** Willkommen zu diesem Programm ! **"
titelende:
even
```

Sie werden diese Funktion sicher öfter benötigen. Zusätzlich kommt es oft vor, daß man nur ein Zeichen ausgeben will. Um daher diese beiden Funktionen abzudecken, bietet sich folgendes Unterprogramm an, welches vier Einsprungpunkte besitzt:

pmsg gibt den Text ab (D2) bis zu einem abschließenden Null-Byte aus.

pline wie oben, nur daß nach dem Text automatisch ein CR ausgeführt wird, also der Cursor an den Anfang der nächsten Zeile gesetzt wird.

pchar gibt das Zeichen in D0 aus.

pcrlf setzt den Cursor an den Anfang der nächsten Zeile.

Hier dieses Unterprogramm-Paket:

```
Write = -48

     ...
pline:                      ;* Zeile ausgeben mit CR
     bsr     pmsg           ;Zeile ausgeben

pcrlf:                      ;* Cursor in die nächste Zeile
     move    #10,d0         ;Linefeed
     bsr     pchar          ;ausgeben
     move    #13,d0         ;und CR

pchar:                      ;* Zeichen in D0 ausgeben
     move.b  d0,outline     ;Zeichen in Ausgabepuffer
     move.l  #outline,d2    ;Adresse des Zeichens

pmsg:                       ;* Zeile ab (D2) bis 0-Byte ausgeben
     move.l  d2,a0          ;Adresse in A0
     clr     d3             ;Länge =0
```

```
ploop:
     tst.b   (a0)+        ;Nullbyte ?
     beq     pmsg2        ;ja: Länge ermittelt
     addq.l  #1,d3        ;sonst Länge+1
     bra     ploop        ;und weitersuchen
pmsg2:
     move.l  dosbase,a6   ;DOS-Basisadresse in A6
     move.l  conhandle,d1 ;unser Fenster-Handle
     jsr     Write(a6)    ;Write-Funktion aufrufen
     rts                  ;fertig!

outline:     dc.w 0       ;Ausgabepuffer für 'pchar'
conhandle:   dc.l 0       ;Handle des Fensters
```

Mit diesem Programm können Sie nun recht einfach beliebige Ausgaben in das CON:-Fenster bringen. Diese Routinen funktionieren auch, wenn Sie ein RAW:-Fenster geöffnet haben, Sie sollten dann allerdings auch den Label 'conhandle' in 'rawhandle' umnennen, damit es nicht evtl. später einmal Verwechslungen geben kann.

Bleiben wir aber zunächst bei den CON:-Fenstern. Wie bereits erwähnt, gibt es für die Ausgabe außer den normalen Zeichen auch noch spezielle Zeichen, welche nicht erscheinen, sondern eine Funktion auslösen oder Parameter für die Ausgabe verändern. Diese Zeichen nennt man Steuerzeichen.

Eines dieser Steuerzeichen haben wir schon kennengelernt, nämlich Linefeed ($A). Dieses Zeichen wird ja auch nicht einfach ausgegeben, sondern löst die Funktion aus, welche den Cursor in die nächste Zeile bringt und ggf. sogar das Hochschieben des Fensterinhaltes bewirkt. Dies ist schon sehr nützlich, aber Sie werden sehen, daß noch viel interessantere Steuerzeichen existieren.

Hier nun eine Liste der Steuerzeichen, welche eine Funktion auslösen. Diese Zeichen sind als Sedezimalzahlen aufgeführt.

Sequenz	Funktion
08	Backspace
0A	Linefeed, Cursor runter
0B	Cursor eine Zeile hoch
0C	Fenster löschen
0D	Carriage Return, Cursor in die erste Spalte
0E	auf Normaldarstellung schalten (0F zurücknehmen)
0F	auf Sonderzeichen schalten
1B	Escape

Die folgenden Sequenzen beginnen mit dem Zeichen $9B, dem CSI (Control Sequence Introducer). Die auf dieses Zeichen folgenden Zeichen bewirken dann eine Funktion. Die in eckigen Klammern angegebenen Werte können entfallen. Die Angabe für 'n' ist in ASCII-Zeichen als ein- oder mehrstellige dezimale Zahl anzugeben. Der Wert, welcher bei weggelassener Angabe für n angenommen wird, ist hier in Klammern hinter n angegeben.

Sequenz	Funktion
9B [n] 40	n Leerzeichen einschieben
9B [n] 41	Cursor um n (1) Zeilen hoch
9B [n] 42	Cursor um n (1) Zeilen runter
9B [n] 43	Cursor um n (1) Zeilen rechts
9B [n] 44	Cursor um n (1) Zeilen links
9B [n] 45	Cursor um n (1) Zeilen runter in Spalte 1
9B [n] 46	Cursor um n (1) Zeilen hoch in Spalte 1
9B [n] [3B n] 48	Cursor in Zeile; Spalte setzen
9B 4A	Fenster ab Cursor löschen
9B 4B	Zeile ab Cursor löschen
9B 4C	Zeile einfügen
9B 4D	Zeile löschen
9B [n] 50	n Zeichen ab Cursor löschen
9B [n] 53	n Zeilen hochschieben
9B [n] 54	n Zeilen runterschieben
9B 32 30 68	ab sofort: Linefeed => Linefeed+Return
9B 32 30 6C	ab sofort: Linefeed => nur Linefeed
9B 6E	sende die Cursor-Position! Es wird eine Zeichenkette folgender Form zurückgegeben: `9B (Zeile) 3B (Spalte) 52`

9B (Stil);(Vordergrundfarbe);(Hintergrundfarbe) 6D
Die drei Parameter sind ebenfalls Dezimalzahlen im ASCII-Format. Sie bedeuten:

Stil: 0 = normal
1 = fett
3 = italic
4 = unterstrichen
7 = invers

Vordergrundfarbe: 30-37
Farben 0-7 für Text

Hintergrundfarbe: 40-47
Farben 0-7 für Hintergrund

9B (Länge) 74 setzt maximale Anzahl der darzustellenden Zeilen fest

9B (Breite) 75 setzt die maximale Zeilenlänge fest

9B (Abstand) 78 definiert den Abstand in Pixeln vom linken Rand des Fensters, ab dem die Ausgabe beginnen soll

9B (Abstand) 79 definiert den Abstand in Pixeln vom oberen Rand des Fensters, ab dem ausgegeben werden soll.

Die letzten 4 Funktionen können durch Weglassen des Parameters auf die Normalstellung gebracht werden.

9B 30 20 70 Cursor unsichtbar machen

9B 20 70 Cursor sichtbar machen

9B 71 sende Fenster-Ausmaße! Es wird eine Zeichenkette folgender Form zurückgegeben:

```
9B 31 3B 31 3B (Zeilen) 3B (Spalten) 73
```

Um die Anwendung dieser Steuerzeichen zu demonstrieren, geben Sie doch einmal folgenden Text mittels der 'pmsg'-Routine in Ihrem Fenster aus:

```
text:  dc.b $9b,"4;31;40m"
       dc.b "Überschrift"
       dc.b $9b,"3;33;40m",$9b,"5;20H"
       dc.b "** Hallo, Welt ! **",0
```

Die Parameter für die Steuersequenzen sind hier einfach durch die Anführungszeichen markiert, als ASCII-Zeichenketten angegeben. Wie Sie sehen, können Sie so einfach recht wirkungsvolle Textausgaben realisieren!

Nachdem wir uns mit der Ausgabe von Zeichen und Texten beschäftigt haben, kommen wir nun zu dessen Gegenstück, nämlich der Eingabe von Texten.

6.4.2 Tastatureingaben

Für die Eingabe von Zeichen über die Tastatur können wir einen einfachen Weg nehmen. Wir brauchen nur aus dem bereits geöffneten Ein-/Ausgabekanal des CON:-Fensters zu lesen. Dafür benötigen wir eine weitere Funktion der DOS-Bibliothek. Diese Funktion heißt READ und trägt den OFFSET -42.

Dieser Funktion werden dieselben drei Parameter wie der WRITE-Funktion übergeben, und zwar:

In D1 die Handle-Nummer, welche wir aus der OPEN-Funktion erhalten haben,

In D2 die Adresse, ab der die gelesenen Daten im Speicher abgelegt werden sollen und

In D3 die Anzahl der zu lesenden Daten.

Hierzu also die übliche Unterroutine, welche soviel Zeichen von der Tastatur bzw. von der Konsole in einen Puffer namens 'inline' einliest, wie in D3 übergeben wird.

```
Read     = -42

     ...
readchr:                    ;* (D3) Zeichen von Tastatur holen
     move.l  #inline,d2         ;Adresse des Puffers in D2
     move.l  dosbase,a6         ;DOS-Basisadresse in A6
     move.l  conhandle,d1       ;unser Fenster-Handle
     jsr     Read(a6)           ;Write-Funktion aufrufen
     rts                        ;fertig!

inline:   blk.b 80,0            ;Puffer für Tastatureingabe
```

Diese Routine kehrt ins Hauptprogramm zurück, wenn die Return-Taste betätigt wurde. Werden mehr als in D3 angegebene Zeichen eingegeben, so werden in 'inline' nur die ersten Zeichen übergeben. Bei einem weiteren Aufruf kehrt die Routine sofort zurück und übergibt den Rest der eingegebenen Zeichen.

Diese Art der Eingabe bietet so manchen Komfort. Es kann bei der Eingabe beliebig mit Backspace gearbeitet werden, da nur die wirklich verbleibende Zeile in 'inline' übergeben wird. Außerdem erhält man die Anzahl der übergebenen Zeichen in D0 zurück.

Probieren Sie dieses Programm einmal folgendermaßen aus:

Schreiben Sie ins Hauptprogramm nach dem Öffnen des CON:-Fensters die Zeilen:

```
     move    #80,d3          ;80 Zeichen zulassen
     bsr     readchr         ;Zeile von Tastatur holen
     lea     inline,a0       ;Adresse der Zeile in A0
     clr.b   0(a0,d0)        ;Nullbyte ans Ende
     bsr     pmsg            ;Zeile wieder ausgeben
bp:
```

Hiernach kommt dann der Programmteil, welcher das Fenster wieder schließt. Wenn Sie nun dieses Programm mit 'g run' starten und als Breakpoint 'bp' angeben, bricht das Programm an dieser Stelle ab. Sie können somit das Ergebnis auf Ihrem Fen-

ster in Ruhe betrachten. Danach können Sie mit 'j bp' das Programm fortführen und das Fenster schließen lassen, wenn das Programm mit einem RTS endet.

Nach dem Start des Programms und dem Öffnen des Fensters erscheint in der oberen linken Ecke des Fensters der Cursor. Geben Sie nun einen kleinen Text ein und drücken Return. Sie sehen, daß in dem Fenster die Zeichenkette, welche Sie gerade eingegeben haben, direkt wieder ausgegeben wird.

Die Ausgabe wurde mit der im vorigen Kapitel vorgestellten 'pmsg'-Routine bewerkstelligt. Diese Routine benötigt eine Null am Ende des auszugebenden Textes. Diese Null wird dadurch an den eingegebenen Text angehängt, daß zuerst die Adresse des Eingabepuffers in A0 geladen und danach das Byte mit CLR.B gelöscht wurde, welches an der Stelle A0+D0 liegt. Da in D0 die Anzahl der eingegebenen Zeichen liegt, ist dieses Byte also das erste unbenutzte Byte.

Da Sie sich ja noch im SEKA befinden, wenn das Programm abbricht, können Sie mit 'q inline' einmal ansehen, was die 'readchr'-Routine geliefert hat. Sie finden dann die Zeichen wieder, die Sie eingetippt haben, plus ein abschließendes $A. Dieses $A steht für die Return-Taste und wird mitgezählt, so daß Sie nach der Eingabe von z.B. '12' und Return als Anzahl eine 3 in D0 bekommen.

Diesen Vorgang sollten Sie nun einmal mit einem RAW:-Fenster wiederholen. Ändern Sie dafür in der Fensterdefinition CON: in RAW: und assemblieren das Programm neu. Sie werden den Unterschied sofort bemerken. Es wird nämlich nach dem ersten Tastendruck sofort wieder zurückgekehrt und demnach in D0 immer eine 1 übergeben.

Der Vorteil dieser Eingabe ist jedoch, daß auch die Cursor- und Funktionstasten erkannt werden. Mit einer geeigneten Routine kann man durch wiederholtes Einlesen eines Zeichens über 'readchr' somit auch diese Sondertasten bearbeiten.

Eine weitere Art der Tastatureingaben ist die Abfrage einer einzelnen Taste. Dies wird z.B. benötigt, wenn ein Programm bei einer Sicherheitsabfrage um die Betätigung der 'J'-Taste für Ja bittet, wenn eine wichtige Funktion auszuführen ist. Dies könnte zwar auch über eine normale Eingabe geschehen, ist jedoch in manchen Fällen auf diese Art besser.

Die DOS-Bibliothek beinhaltet eine Funktion, die genau dies macht. Sie wartet eine vorgegebene Zeit auf die Betätigung einer Taste und gibt dann entweder eine Null (FALSE) zurück, wenn in dieser Zeit keine Taste gedrückt wurde, oder den Wert -1 ($FFFFFFFF = TRUE), wenn doch. Welche Taste betätigt wurde, muß nachher extra ausgewertet werden. Diese Funktion, die den Namen WaitForChar trägt, ist somit alleine nur sinnvoll, wenn z.B. bei der Ausgabe eines längeren Textes nach einer Seite gestoppt und auf eine Taste gewartet werden soll, um die nächste Seite anzuzeigen.

Die Funktion benötigt nur zwei Parameter:

In D1 die Handle-Nummer des Fensters bzw. der Datei, aus der das Zeichen gelesen werden soll. Es kann somit auch auf ein Zeichen von einer Schnittstelle gewartet werden.

In D2 wird die Zeit in Mikrosekunden übergeben, die auf einen Tastendruck bzw. den Empfang eines Zeichens gewartet werden soll.

Um also auf die Eingabe einer Taste in unserem Fenster eine Sekunde zu warten, müssen wir folgende Routine schreiben:

```
WaitForCh=-30-174

     ...
scankey:                        ;* Warten auf Tastendruck
     move.l  conhandle,d1       ;in unserem Fenster
     move.l  #1000000,d2        ;Wartezeit 1 Sekunde
     move.l  dosbase,a6         ;DOS-Basisadresse
     jsr     waitforch(a6)      ;warten...
     tst.l   d0                 ;Ergebnis testen
     rts
```

Durch den TST-Befehl am Ende des Programms können Sie nach dem Aufruf dieser Routine direkt mit einem BEQ- bzw. BNE-Befehl das Ergebnis der Routine auswerten: BEQ verzweigt, wenn keine Taste gedrückt wurde, BNE nicht.

6.4.3 Drucker-Ansteuerung

Nachdem wir die Ein- und Ausgaben über die Konsole besprochen haben, wollen wir nun auch Daten aus dem Rechner hinausgeben. Das erste Gerät, welches sich dazu anbietet, ist der Drucker.

Ausgaben auf dem Drucker sind sehr einfach zu realisieren. Sie müssen nur einen weiteren Kanal öffnen, wie Sie es von Fenstern her schon gewöhnt sind. Lediglich als Name wird hier nicht CON: oder RAW: angegeben, sondern PRT:.

Sie öffnen also diesen Kanal mit den gleichen Zeilen wie vorher das Fenster, nur daß Sie hier einen Zeiger auf den Kanalnamen "PRT:" in D1 und den Modus 'neu' (1006) in D2 an die 'do_open'-Routine übergeben. Die von dort zurückgegebene Handle-Nummer speichern Sie dann irgendwo ab, z.B. an dem Label 'prthandle'.

Nun können Sie die gleichen Ausgaberoutinen wie die für die Fenster verwenden, um Zeichen bzw. Texte auf dem Drucker auszugeben. Sie müssen lediglich anstelle von 'conhandle' den Namen 'prthandle' in die Zeile 'move.l conhandle,d1' eintragen

Noch besser wäre es, diese Zeile ganz zu streichen. Dann können Sie nämlich wirklich die gleichen Routinen verwenden, nur daß für Fensterausgaben der Wert aus 'conhandle' und für Druckerausgaben der aus 'prthandle' in D1 an die Routine übergeben werden muß. Auf diese Weise haben Sie eine recht flexible Ausgaberoutine, welche auf gleiche Art und Weise sowohl das Fenster wie auch den Drucker bedient. Eine Eingabe vom Drucker wird natürlich nicht funktionieren, da der Drucker ja keine Daten senden, sondern nur empfangen und drucken kann.

6.4.4 Serielle Ein-/Ausgabe

Die serielle Schnittstelle läßt sich genauso einfach wie der Drucker bedienen, wenn als Filename SER: angegeben wird. Der damit geöffnete Kanal läßt nun normale Ein-/Ausgaben zu, die mit den DOS-Funktionen READ und WRITE vorgenommen werden. Die Parameter für die Schnittstelle wie Handshake oder Übertragungsgeschwindigkeit lassen sich mit dem Preferences-Programm einstellen.

6.4.5 Sprachausgabe

Der Amiga hat als Besonderheit einen Sprachsynthesizer eingebaut. Dies zu programmieren ist leider nicht so einfach wie bei den bisher besprochenen Ein-/Ausgaben. Es handelt sich hierbei um ein sogenanntes Device mit dem Namen 'narrator.device'.

Diese Devices erfordern einige Programmierschritte, um sie zu installieren und anzusprechen. Hierzu gehört das Öffnen des Devices, Starten der Ein-/Ausgabe usw. Ich möchte hier auf all diese Dinge nicht in allen Einzelheiten eingehen, sondern Ihnen eine Routine vorstellen, welche einen Text sowohl vom normalen Text in die notwendige Lautschrift übersetzt als auch ausgibt.

Zuerst brauchen wir dafür einige Initialisierungen. Dazu gehört die übliche Definition der Konstanten, zu denen einige noch unbekannte beigefügt werden. Hier die wichtigsten, bei denen das Öffnen eines Fensters ebenfalls vorgesehen ist:

```
;***** Narrator-Grundfunktionen 3/87 S.D. *****

OpenLib  =-408
closelib =-414
ExecBase =4

Open     =-30      ;File öffnen
Close    =-36      ;File schließen
mode_old =1005     ;Modus: alt
```

```
OpenDevice=-444     ;Device öffnen
CloseDev  =-450     ;Device schließen

SendIo    =-462     ;I/O starten
AbortIO   =-480     ;I/O abbrechen

Translate =-30      ;Text übersetzen
```

Danach folgen die Initialisierungen selbst:

```
init:                           ;System initialisieren und öffnen

;* DOS-Bibliothek öffnen *

    move.l  execbase,a6         ;Zeiger auf EXEC-Bibliothek
    lea     dosname,a1          ;Zeiger auf DOS-Namen
    moveq   #0,d0               ;Version: egal
    jsr     openlib(a6)         ;DOS-Library öffnen
    move.l  d0,dosbase          ;Handle retten
    beq     error               ;ggf. Fehler

;* Übersetzer-Bibliothek (translator.library) öffnen

    lea     transname,a1        ;Zeiger auf Translator-Namen
    clr.l   d0
    jsr     openlib(a6)         ;Translator öffnen
    move.l  d0,tranbase         ;Handle retten
    beq     error               ;ggf. Fehler

;* I/O-Bereich für Narrator setzen *

    lea     talkio,a1           ;Zeiger auf I/O-Bereich in A1
    move.l  #nwrrep,14(a1)      ;Portadresse eintragen
    move.l  #amaps,48+8(a1)     ;Zeiger auf Audiomasken
    move    #4,48+12(a1)        ;Anzahl der Masken
    move.l  #512,36(a1)         ;Länge des Ausgabebereichs
    move    #3,28(a1)           ;Kommando: Schreiben
    move.l  #outtext,40(a1)     ;Adresse des Ausgabebereiches

;* Narrator-Device öffnen *

    clr.l   d0                  ;Nummer 0
    clr.l   d1                  ;Flags: keine
    lea     nardevice,a0        ;Zeiger auf Device-Namen
```

```
        jsr     opendevice(a6)   ;Narrator.device öffnen
        tst.l   d0               ;Fehler ?
        bne     error            ;ja!

;* Fenster öffnen *

        move.l  #consolname,d1   ;Consol-Definition
        move.l  #mode_old,d2     ;Modus: alt
        move.l  dosbase,a6       ;DOS-Basisadresse
        jsr     Open(a6)         ;Fenster öffnen
        tst.l   d0               ;Fehler ?
        beq     error            ;ja!
        move.l  d0,conhandle     ;sonst Handle retten
```

Nachdem wir diese Initialisierungen vorgenommen haben, können wir nun den vorbereiteten Text aufsagen lassen. Um auch zu sehen, was der Amiga uns da erzählt, können wir mit Hilfe der 'pline'-Routine diesen Text gleich im Fenster ausgeben lassen:

```
        move.l  #intext,d2       ;zu sprechenden Text
        bsr     pline            ;auch im Fenster ausgeben

sayit:                           ;Text sprechen lassen

;* Text in Lautschrift übersetzen *

        lea     intext,a0        ;Adresse des Textes
        move.l  #outtext-intext,d0 ;Länge des Textes
        lea     outtext,a1       ;Adresse des Ausgabebereiches
        move.l  #512,d1          ;Länge des Ausgabebereiches
        move.l  tranbase,a6      ;Translator-Basisadresse
        jsr     Translate(a6)    ;Text übersetzen

;* Sprachausgabe *

        lea     talkio,a1        ;Adresse der I/O-Struktur
        move.l  #512,36(a1)      ;Länge des Ausgabebereiches
        move.l  execbase,a6      ;EXEC-Basisadresse
        jsr     SendIO(a6)       ;I/O bzw. Sprachausgabe starten
```

Da wir bei Beendigung des Programms die Ein-/Ausgabe stoppen, müssen wir nun etwas einfügen, was das Programm 'anhält'. Dafür bietet sich die 'readchr'-Funktion an, welche wir ja bereits programmiert haben:

```
        bsr     readchr         ;Warten auf Tastatureingabe
```

Der Rechner wartet jetzt so lange, bis wir die Return-Taste betätigen. Wir können uns also in aller Ruhe anhören, was der Amiga uns zu sagen hat. Wird dann Return gedrückt, so wird alles wieder ausgeschaltet bzw. geschlossen:

```
quit:
        move.l  execbase,a6     ;EXEC-Basisadresse
        lea     talkio,a1       ;Zeiger auf I/O-Bereich
        jsr     abortio(a6)     ;I/O abbrechen

        move.l  conhandle,d1
        move.l  dosbase,a6
        jsr     close(a6)       ;Fenster schließen

        move.l  dosbase,d1
        move.l  execbase,a6
        jsr     closelib(a6)    ;DOS-Bibliothek schließen

        lea     talkio,a1
        jsr     closedev(a6)    ;Narrator.device schließen

        move.l  tranbase,a1
        jsr     closelib(a6)    ;Translator-Bib. schließen

        rts                     ;* Ende des Programms
```

Nun folgen noch die notwendigen Daten, die wir für obiges Programm brauchen:

```
text:       dc.b  'Dies ist ein Test-Text !',10,13,10,13,0,0
dosname:    dc.b  'dos.library',0,0
transname:  dc.b  "translator.library",0
consolname: dc.b  'RAW:0/100/640/100/** Test-Fenster',0
nardevice:  dc.b  'narrator.device',0
even
dosbase:    dc.l  0
tranbase:   dc.l  0
```

```
amaps:      dc.b  3,5,10,12
even
conhandle:  dc.l  0
talkio:     blk.l 20,0
nwrrep:     blk.l 8,0
intext:     dc.b  'hello, i am the amiga talking to you',0
even
outtext:    blk.b 512,0
```

Dies alles ist einigermaßen umständlich, wird aber durch die dadurch eröffneten Möglichkeiten reichlich belohnt. Es gibt hier nämlich noch eine Menge Variationen, die wir durch die Änderung einiger Parameter erhalten. Diese Parameter sind die Eintragungen in den I/O-Bereich, welcher ab dem Label 'talkio' liegt. Dieser Bereich ist folgendermaßen aufgebaut:

Offset	Länge	Bedeutung
** Port-Daten **		
0	L	Zeiger auf nächsten Block
4	L	Zeiger auf letzten Block
8	B	I/O-Typ
9	B	Priorität
10	L	Zeiger auf I/O-Namen
14	L	Zeiger auf Port
18	W	Länge
** I/O-Daten **		
20	L	Zeiger auf Device
24	L	Zeiger auf Device-Unit
28	W	Kommandowort
30	B	I/O-Flags
31	B	I/O-Status
32	L	I/O-Zeiger
36	L	I/O-Länge
40	L	Zeiger auf Daten
44	L	I/O-Offset
** eigentliche Narrator-Daten **		
48	W	Sprechgeschwindigkeit
50	W	Stimmhöhe
52	W	Sprachmodus

54	W	Geschlecht (männliche/weibliche Stimme)
56	L	Zeiger auf Audio-Masken
60	W	Anzahl der Masken
62	W	Lautstärke
64	W	Abtastrate
66	B	Flag für Mundgraphik-Erzeugung (0=aus)
67	B	aktuelle Maske (interne Verwendung)
68	B	verwendete Kanäle (interne Verwendung)

Es ist nicht empfehlenswert, mit den Daten der ersten zwei Blöcke zu experimentieren, da die leicht mit einem Guru enden kann. Interessant sind dagegen die letzten Einträge dieser Struktur, mit denen wir viele interessante Effekte erzielen können.

Hier eine Übersicht über die Parameter, mit denen wir die Sprachausgabe variieren können. Der in Klammern angegebene Wert stellt den Standard-Wert dar, welcher schon beim Öffnen des Narrator.device eingesetzt wird.

Sprechgeschwindigkeit (150)

Hiermit können Sie die Geschwindigkeit einstellen, mit der gesprochen wird. Die Stimmlage bleibt dabei weitgehend unverändert.

Stimmhöhe (110)

Hier können Sie einen Wert zwischen 65 und 320 eintragen und so die Stimmlage verändern (von Goofy bis Mickey Mouse).

Sprachmodus (0)

Die Null bewirkt hier eine halbwegs natürliche Aussprache. Eine Eins läßt den Amiga monoton wie einen Roboter sprechen.

Geschlecht (0)

Hier wird durch eine Null eine männliche und durch eine Eins eine weibliche Stimmlage eingestellt (mehr oder weniger...).

Lautstärke (64)

Die Lautstärke kann zwischen 0 und 64 eingestellt werden. Die Grundlautstärke ist somit maximal.

Abtastrate (22200)

Durch die Senkung dieses Wertes wird die Stimme ebenfalls gesenkt. Bei starken Änderungen entstehen recht eigenartige Stimmen!

Sie können nun einmal ein wenig herumexperimentieren, bis Sie die interessanteste Stimme gefunden haben. Wenn Sie dann jedoch einen deutschen Text aussprechen lassen wollen, so werden Sie feststellen, daß der Amiga einen scheußlichen Dialekt hat!

Aus diesem Grund muß man, um einen deutschen Text aufsagen zu lassen, diesen Text in 'amerikanischer Schreibweise' eingeben. Als Beispiel:

Um zu hören: 'Dies ist ein deutscher Text', muß man schreiben: 'dees ist hine doytsher text'

Dies sieht zwar merkwürdig aus, klingt aber durchaus verständlich. Probieren Sie es aus!

6.5 Diskettenoperationen

Das wichtigste Peripheriegerät für einen Computer wie den Amiga ist das Diskettenlaufwerk. Hierauf werden alle Daten gerettet, die nach dem Ausschalten des Rechners nicht verloren gehen sollen. Wie man dieses Retten und natürlich auch das Wiederholen der Daten programmieren kann, soll in diesem Kapitel besprochen werden.

Wir beschäftigen uns zuerst einmal mit den einfachen Diskettenoperationen, welche zur Verwaltung von Dateien dienen. Um an eine solche Datei heranzukommen, müssen wir sie erst öffnen. Diese Operation ist uns bereits eingehend bekannt, da es sich hierbei um dieselbe OPEN-Funktion der DOS-Bibliothek han-

delt, wie wir sie bereits im vorigen Kapitel mehrmals verwendet haben. Ich gehe in den folgenden Beispielen davon aus, daß Sie bereits die DOS-Bibliothek geöffnet haben.

6.5.1 Dateien öffnen

Bei dieser OPEN-Funktion haben wir festgestellt, daß hier ein Parameter für den Modus erforderlich ist. Dieser Modus hat hier eine besondere Bedeutung. Wird die Datei geöffnet, um aus ihr zu lesen, ist sie scheinbar schon vorhanden. Der Modus muß daher mit 'alt' (1005) angegeben werden.

Will man eine Datei anlegen, muß man sie ebenfalls zuerst öffnen. Da sie dann wohl noch nicht existiert, wird der Modus 'neu' (1006) übergeben. Wird eine Datei zum Schreiben mit diesem Modus geöffnet, obwohl sie unter dem Namen bereits existiert, so wird die alte Datei dieses Namens zuerst gelöscht und danach wieder neu angelegt. Man sollte daher zur Vermeidung von Datenverlusten immer vorher prüfen, ob es die Datei schon gibt und dann ggf. eine Fehlermeldung ausgeben.

Wir legen uns also zuerst eine Unterroutine an, die eine Datei öffnet. Wir gehen dabei davon aus, daß der Dateiname ab dem Label 'filename' steht und mit einem Nullbyte abgeschlossen ist. Wir brauchen dieser Unterroutine dann nur noch den Modus im Register D2 zu übergeben.

Die Routine legt die erhaltene Filehandle-Nummer dann in 'filehd' ab und kehrt ins aufrufende Hauptprogramm zurück. Da die Operation mit dem Handle die letzte des Unterprogramms ist, kann nach der Rückkehr direkt der Status der Operation ausgewertet werden: Ist die Operation korrekt ausgeführt worden und die Datei geöffnet, so ist auch die Handle-Nummer ungleich Null. Andernfalls ist sie Null, was uns also die Möglichkeit gibt, nach dem 'bsr openfile' direkt mit 'beq error' in eine Routine zu verzweigen, welche evtl. eine Fehlermeldung ausgibt.

Hier nun das Unterprogramm zum Öffnen einer Datei und gleich noch eines zum Schließen dieser Datei:

```
Open     = -30
Close    = -36
Mode_old = 1005
Mode_new = 1006

     ...
openfile:                    ;* Datei öffnen, Modus in D0
     move.l  dosbase,a6      ;DOS-Basisadresse in A6
     move.l  filename,d1     ;Zeiger auf Dateinamen
     jsr     Open(a6)        ;Datei öffnen
     move.l  d0,filehd       ;Handle retten
     rts

closefile:                   ;* Datei schließen
     move.l  dosbase,a6      ;DOS-Basisadresse in A6
     move.l  filehd,d1       ;Filehandle in D1
     jsr     Close(a6)       ;Datei schließen
     rts

filehd:   dc.l  0            ;Platz für Filehandle
filename: dc.b  "Dateiname",0 ;zu öffnende Datei
even
```

Um diese Routinen anzuwenden, müssen wir uns nun damit beschäftigen, wie wir Daten laden und abspeichern können.

6.5.2 Daten lesen oder schreiben

Beginnen wir mit dem Anlegen und Beschreiben einer neuen Datei. Um damit zu beginnen, schreiben wir zuerst die Zeilen:

```
     move.l  #Mode_new,d2    ;neue Datei anlegen
     bsr     openfile        ;Datei öffnen
     beq     error           ;hat nicht geklappt!
```

Als Dateiname schreiben wir in die Zeile mit 'filename' irgendeinen Namen, z.B. "Testfile", der mit einem Nullbyte abgeschlossen wird. Nach dem Aufruf der 'openfile'-Routine wird eine Datei diesen Namens auf der Diskette angelegt; sollte sie schon existiert haben, so wird sie zuerst gelöscht.

Nehmen wir nun an, wir haben einen kleinen Text, welchen wir in diese Datei schreiben wollen. Für unser Beispielprogramm können wir ja einen Text vorgeben, indem wir schreiben:

```
text:     dc.b "Dies ist ein Test-Text für das Testfile",0
textende:
```

Der Label 'textende' dient dazu, durch eine Subtraktion von 'text' die Anzahl der Datenbytes zu ermitteln.

Diesen Text wollen wir nun in die Datei schreiben. Dafür verwenden wir wieder die ebenfalls bekannte Funktion WRITE, welche drei Parameter braucht:

In D1 das Filehandle, welches wir aus der OPEN-Funktion erhalten haben,

In D2 einen Zeiger auf die Daten, welche geschrieben werden sollen und

In D3 die Anzahl der zu schreibenden Daten in Bytes.

Für unser Beispiel bedeutet dies, daß wir ein weiteres Unterprogramm schreiben können, welcher wir den Zeiger auf die Daten in D2 und die Anzahl der Daten in D3 übergeben müssen:

```
Write     = -48

     ...
writedata:                ;* Daten in die Datei schreiben
     move.l  dosbase,a6   ;DOS-Basisadresse in A6
     move.l  filehd,d1    ;Filehandle in D1
     jsr     Write(a6)    ;Daten schreiben
     rts
```

Diese Unterroutine rufen wir dann auf, indem wir ins Hauptprogramm nach dem Öffnen der Datei folgende Zeilen schreiben:

```
     move.l  #text,d2           ;Zeiger auf die Daten
     move.l  #textende-text,d3  ;Anzahl der Bytes
     bsr     writedata          ;Daten in die Datei schreiben
```

Danach schließen wir die Datei mit

```
bsr     closefile       ;Datei schließen
bra     ende            ;Programm beenden
```

Wenn Sie nun mit dem SEKA dieses Programm assemblieren, starten und nachher mit 'v' das Directory auslesen, finden Sie dort die neue Datei 'Testfile', welche die Länge unseres Textes hat. Diese Datei wollen wir als nächstes wieder einladen, um festzustellen, ob sie auch die richtigen Daten enthält.

Dafür nehmen wir die READ-Funktion des DOS, welche die gleichen Parameter wie WRITE benötigt. Die Angabe der Anzahl der zu lesenden Bytes kann dafür verwendet werden, nur einen Teil der Datei einzulesen. Gibt man hier mehr an, als in der Datei enthalten ist, so wird die gesamte Datei geladen. Die Anzahl der gelesenen Bytes finden Sie dann im Register D0.

Wir legen nun ein Feld an, in dem genug Platz für die zu lesenden Daten ist. Dies können wir folgendermaßen machen:

```
feld:   blk.b  100          ;100 Bytes reservieren
```

Für unsere Beispieldatei reicht dies völlig aus. Wenn Sie eine andere Datei laden wollen, müssen Sie hier natürlich mehr Platz reservieren.

Für das Lesen der Datei schreiben wir nun eine weitere Unterroutine. Dabei gehen wir davon aus, daß immer die gesamte Datei geladen werden soll. Wir brauchen dann der Unterroutine nur die Adresse des Puffers anzugeben,in den die Daten geladen werden sollen, in unserem Beispiel die Adresse 'feld'.

Hier nun die Unterroutine, die die geöffnete Datei vollständig in den Speicherbereich läd, der in D2 angegeben wird:

```
Read      = -42

     ...
readdata:                        ;* Datei auslesen
     move.l  dosbase,a6          ;DOS-Basisadresse in A6
     move.l  filehd,d1           ;Filehandle in D1
     move.l  #$ffffff,d3         ;beliebig viele Bytes lesen
     jsr     Read(a6)            ;Daten lesen
     rts
```

Um nun mit diesen Routinen die eben angelegte Datei in den unter 'feld' angelegten Puffer zu laden, verwenden wir folgendes Hauptprogramm:

```
     move.l  #Mode_old,d2        ;alte Datei
     bsr     openfile            ;Datei öffnen
     beq     error               ;hat nicht geklappt !

     move.l  #feld,d2            ;Zeiger auf Datenpuffer
     bsr     readdata            ;Datei einlesen
     move.l  d0,d6               ;Byteanzahl in D6 retten

     bsr     closefile           ;Datei schließen
     bra     ende                ;Programmende
```

Nach der Assemblierung und dem Starten dieses Programms können Sie sich nun mit 'q feld' im Datenpuffer ansehen, was Sie aus der Datei eingelesen haben. In D6 finden Sie außerdem die Anzahl der Bytes, welche aus der Datei gelesen wurden.

6.5.3 Dateien löschen

Wenn Sie nun genug mit obigen Programmen experimentiert haben, möchten Sie sicher die Datei 'Testfile' wieder von der Diskette löschen. Hierfür gibt es die Funktion DELETEFILE in der DOS-Bibliothek, welche den Offset -42 hat und nur einen Parameter benötigt. Dieser Parameter wird in D1 übergeben und stellt einen Zeiger auf den Dateinamen dar. Der Dateiname muß natürlich wieder mit einem Nullbyte abgeschlossen werden.

Um also die Datei mit dem Namen 'Testfile' zu löschen, können Sie folgende Zeilen schreiben:

```
DeleteFile  = -42

        ...
        move.l  dosbase,a6        ;DOS-Basisadresse in A6
        move.l  #filename,d1      ;Zeiger auf Dateinamen in D1
        jsr     DeleteFile(a6)    ;Datei löschen
```

Weg ist die Datei. Diese Datei ist auch nicht mehr mit normalen Mitteln zu retten, falls die falsche Datei gelöscht wurde! Wir sollten daher bei wichtigen Dateien evtl. einen Trick anwenden, um die alte Datei zu retten. Diesen Trick wenden viele Programme auch an.

6.5.4 Dateien umbenennen

Wenn z.B. ein Textverarbeitungsprogramm einen Text, welcher gerade geändert wurde, wieder auf die Diskette zurückschreibt, so wird die alte Datei selten gelöscht. Oft wird nämlich diese alte Datei zuerst umbenannt, z.B. bekommt sie den Namen 'Backup', und dann der neue Text in die dann neue Datei des alten Namens auf Diskette geschrieben.

Die Funktion der DOS-Bibliothek, welche diese Umbenennung von Dateien bewerkstelligt, heißt RENAME und hat den Offset -48. Ihr werden als Parameter zwei Zeiger übergeben, und zwar in D1 den Zeiger auf den alten und in D2 den Zeiger auf den neuen Namen der Datei.

Um also die Datei 'Testfile' (natürlich vor dem Löschen) in 'Backup' umzutaufen, verwenden wir folgende Zeilen:

```
Rename    = -48

        ...
        move.l  dosbase,a6        ;DOS-Basisadresse in A6
        move.l  #oldname,d1       ;Zeiger auf alten Namen in D1
        move.l  #newname,d2       ;Zeiger auf neuen Namen in D2
        jsr     Rename(a6)        ;Datei umbenennen
```

```
        ...
oldname:  dc.b  "Testfile",0
newname:  dc.b  "Backup",0
```

6.5.5 Inhaltsverzeichnis auslesen

Stellen Sie sich vor, Sie haben einen Text-Editor programmiert und dieses Programm gestartet. Nun wollen Sie einen Text von der Diskette laden und bearbeiten - doch wie hieß denn noch diese Textdatei?

Was wir also brauchen, ist eine Funktion zum Auslesen und Anzeigen des Inhaltsverzeichnisses der Diskette. Dafür gibt es sogar mehrere Wege, um dies zu bewerkstelligen. Betrachten wir zuerst den einfacheren Weg, welcher zwar wenig Programmieraufwand, jedoch eventuell zwei Diskettenlaufwerke bzw. Diskettenwechsel erfordert.

Der ganze Trick dabei ist, einfach das im C-Directory enthaltene Dir- oder List-Programm bzw. das entsprechende Kommando des CLI selbst aufzurufen. Die DOS-Bibliothek beinhaltet dafür ein sehr einfaches Kommando namens 'Execute' mit dem Offset -222, welche es ermöglicht, ein CLI-Kommando ausführen zu lassen.

Diese Funktion bekommt drei Parameter mit auf den Weg:

In D1 einen Zeiger auf den mit Null abgeschlossenen Text, welcher das auszuführende Kommando angibt. Der Text entspricht genau der Anweisung, welche man in den CLI eintippen würde. Auch ein Null- Zeiger ist hier möglich.

In D2 wird die Eingabe-Datei bestimmt. Normalerweise wird hier eine Null eingetragen. Gibt man jedoch hier z.B. das File-Handle einer geöffneten Text- Datei an, so wird diese Datei nach der Ausführung des vorgegebe-

nen Kommandos ausgelesen und als Kommandosequenz interpretiert. Auch ein Fenster kann als Eingabemedium definiert werden, wodurch man ein neues CLI-Fenster erhält!

In D3 steht die Angabe der Ausgabe-Datei. Ist hier eine Null angegeben, so werden die Ausgaben des aufgerufenen Kommandos (z.B. DIR-Ausgaben) im Standard-Fenster erscheinen (CLI-Fenster).

Um dies einmal auszuprobieren, geben Sie bitte folgendes Unterprogramm in das Hauptprogramm ein, in dem die DOS-Bibliothek und ein Fenster bereits geöffnet wurden:

```
Execute  = -222

     ...
dir:
     move.l  dosbase,a6        ;DOS-Basisadresse in A6
     move.l  #command,d1       ;Zeiger auf Kommandozeile
     clr.l   d2                ;keine Eingaben (CLI-Fenster)
     move.l  windowhd,d3       ;Ausgaben in unserem Fenster
     jsr     Execute(a6)       ;Kommando ausführen
     rts

command:
     dc.b  "dir",0
```

Dieses Programm funktioniert natürlich auch mit dem List-Kommando. Der Nachteil dieser Methode ist allerdings der, daß entweder die Diskette, mit der die Workbench geladen wurde, in einem Laufwerk stecken muß, oder dementsprechend nach der Aufforderung des Systems diese Diskette eingelegt werden muß. Das Dir-Kommando ist ja schließlich nichts anderes als ein Programm, welches der Amiga vor der Ausführung erst einmal laden muß.

Die Vorteile obiger Methode sind allerdings auch nicht zu verachten, ist das Programm doch schön kurz und bietet außerdem die Möglichkeit, jedes beliebige Kommando des CLI in ein Programm einzubauen!

Wir wollen die Vorteile jedoch nun außer acht lassen und uns eine Methode einfallen lassen, die das CLI nicht benötigt. Dadurch können wir dann aus jeder beliebigen Diskette das Inhaltsverzeichnis auslesen, ohne Disk-Jockey zu spielen.

Was wir also programmieren müssen, ist all das, was das Dir-Programm des CLI auch macht. Hierbei handelt es sich um mehrere Schritte:

Zuerst müssen wir dem System einen Schlüssel zu dem gewünschten Directory geben. Dies bedeutet, daß wir die Lock-Funktion des DOS aufrufen müssen, welcher wir zwei Parameter übergeben:

In D1 übergeben wir ein Zeiger auf einen Text, welcher das einzulesende Inhaltsverzeichnis bestimmt. Dieser Text entspricht dem der Dir-Funktion, wenn wir z.B. den Inhalt der RAM-Disk haben wollen, lautet er 'RAM:',0.

In D2 steht der Modus, welcher bestimmt, ob gelesen oder geschrieben werden soll. Wir nehmen hierfür den Modus 'Lesen' (-2).

Damit rufen wir die Lock-Funktion mit dem Offset --84 auf und erhalten entweder einen Zeiger auf den Schlüssel oder eine Null in D0 zurück. Bei einer Null war der Aufruf erfolglos bzw. wurde die Datei nicht gefunden. Mit dieser Funktion können wir also auch leicht feststellen, ob eine Datei auf der Diskette vorhanden ist, indem wir einfach deren Namen übergeben und testen, ob in D0 eine Null zurückkommt. Wenn nicht, so existiert die Datei.

Gehen wir davon aus, daß die Datei bzw. der Pfad existiert. Der erhaltene Wert in D0 ist nun abzuspeichern. Dieser wird nämlich von den beiden Funktionen benötigt, welche wir noch aufrufen müssen.

Die nun folgende Funktion trägt den Namen Examine. Mit ihr wird die Diskette nach dem ersten passenden Eintrag durchsucht und die zu diesem Eintrag gehörenden Parameter wie Name, Länge oder Datum zurückgegeben. Für diese Informationen

müssen wir einen ausreichenden Speicherbereich reservieren und dessen Anfang der Examine-Funktion in D2 übergeben. In D1 übergeben wir den Schlüssel, welchen die Lock-Funktion geliefert hat.

Der Speicherbereich, welcher so mit Informationen gefüllt wird, nennt sich FileInfoBlock. Er ist 260 Bytes groß und enthält alle Informationen über die Datei. Der Name liegt dabei ab dem 9. Byte und endet mit einem Null-Byte, so daß wir ihn leicht mit unserer 'pmsg'-Routine ausgeben können. Die Informationen, die Examine ausgibt, sind allerdings nicht diejenigen über irgendeine Datei, sondern über die Diskette. Der Name im FileInfoBlock wird somit der Diskettenname sein.

Die Examine-Funktion gibt in D0 den Status zurück, welcher angibt, ob die Datei gefunden wurde. Da die vorangegangene Lock-Funktion dies jedoch schon getestet hat, ist die Auswertung dieses Status eigentlich unnötig.

Nun zu der Funktion, mit der wir die einzelnen Dateien aus dem Inhaltsverzeichnis auslesen können. Diese Funktion trägt den bezeichnenden Namen ExNext, was soviel wie Examine Next heißt. Es wird mit dieser Funktion also bei jedem Aufruf der nächste Eintrag gesucht, auf den der Schlüssel paßt. Als Parameter bekommt ExNext die selben Daten wie Examine übergeben. Der Rückgabeparameter in D0 spielt hierbei jedoch eine größere Rolle als vorher.

Die ExNext-Funktion wird immer wieder auf die gleiche Weise aufgerufen und liefert dann immer den nächsten Eintrag des Directory. Ist jedoch kein weiterer passender Eintrag vorhanden, so liefert ExNext in D0 eine Null.

Somit wären wir fertig mit dem Inhaltsverzeichnis. Um jedoch sauber abzuschließen, sollten wir noch nachsehen, ob der Status der letzten Operation auch wirklich 'kein weiterer Eintrag' lautet. Dies können wir durch die IoErr-Funktion des DOS erfahren.

Diese Funktion benötigt keine Parameter. Sie gibt uns in D0 den Status der zuletzt ausgeführten I/O-Operation zurück. Nach dem letzten ExNext muß dieser 232 lauten, was no_more_Entries bedeutet.

Hier nun eine vollständige Routine, welche das Inhaltsverzeichnis der Diskette in Laufwerk DF0: in dem Fenster ausgibt, dessen Handle in 'windowhd' steht, und nachher den Status (232=$E8) in D6 zurückgibt:

```
Lock      = -30-54
Examine   = -30-72
ExNext    = -30-78
IoErr     = -30-102

     ...
directory:                    ;* Inhaltsverzeichnis von DF0:
     move.l  #titel,d0
     bsr     pmsg             ;Überschrift ausgeben

     move.l  dosbase,a6       ;DOS-Basisadresse in A6
     move.l  #name,d1         ;Zeiger auf Pfad-/Dateiname
     move.l  #-2,d2           ;Modus 'Lesen'
     jsr     Lock(a6)         ;Datei suchen
     tst.l   d0               ;gefunden ?
     beq     error            ;nein!
     move.l  d0,locksav       ;sonst Schlüssel retten

     move.l  dosbase,a6       ;DOS-Basisadresse
     move.l  locksav,d1       ;Schlüssel in D1
     move.l  #fileinfo,d2     ;Zeiger auf FileInfoBlock
     jsr     Examine(a6)      ;Disk-Namen holen
     tst.l   d0               ;OK ?
     beq     error            ;nein (kommt kaum vor)
     bra     ausgeben         ;sonst Namen ausgeben

loop:                         ;* Dateinamen auslesen
     move.l  dosbase,a6       ;DOS-Basisadresse
     move.l  locksav,d1       ;Schlüssel in D1
     move.l  #fileinfo,d2     ;Zeiger auf FileInfoBlock
     jsr     ExNext(a6)       ;nächste Datei suchen
     tst.l   d0               ;gefunden ?
     beq     error            ;nein: Ende
```

```
ausgeben:                       ;* Namen ausgeben
     move.l  #fileinfo+8,d0     ;Zeiger auf Namen
     bsr     pline              ;Namen ausgeben
     bra     loop               ;und weitermachen...

error:                          ;* I/O-Status ermitteln
     move.l  dosbase,a6         ;DOS-Basisadresse in A6
     jsr     IoErr(a6)          ;Status holen
     move.l  d0,d6              ;und in D6 retten
     rts                        ;Ende...


name:     dc.b  'DF0:',0
even
locksav:  blk.l   0
fileinfo: blk.l 260
```

Die Einträge, welche in den FileInfoBlock eingetragen werden, sind folgende:

Offset	Name	Bedeutung
0	DiskKey.L	Diskettennummer
4	DirEntryType.L	Eintragstyp (+=Directory, -=Datei)
8	FileName	108 Bytes mit dem Dateinamen
116	Protection.L	Datei geschützt?
120	EntryType.L	Eintragstyp
124	Size.L	Dateilänge in Bytes
128	NumBlocks.L	Anzahl der damit belegten Blocks
132	Days.L	Erstellungsdatum
136	Minute.L	Erstellungszeit
140	Tick.L	Erstellungszeit
144	Comment	116 Bytes mit Kommentar

Wenn Sie also in obigem Programm zusätzlich zum Namen noch die Dateilänge ausgeben wollen, so können Sie dies durch Auslesen der Länge mit 'move.l fileinfo+124,d0' und folgendem Aufruf der Umwandlungsroutine bewerkstelligen, welche daraus eine Dezimalzahl macht. Deren Ergebnis können Sie dann zusammen mit dem Namen ausgeben lassen.

6.5.6 Direkter Zugriff auf Diskette

Der direkte Zugriff auf einzelne Sektoren der Diskette ist nicht als einfache Funktion in den Bibliotheken enthalten. Wir müssen hierfür, ebenso wie bei der Sprachausgabe, wieder einmal mit einem Device arbeiten. Diesmal heißt dieses Device Trackdisk.device.

Wir wollen nun mit diesem Device die direkte Programmierung des oder der Laufwerke ausprobieren. Wenn wir erst einmal ein Programmgerüst aufgebaut haben, können wir die einzelnen Kommandos zum Diskettenzugriff ausprobieren. Beachten Sie bitte dabei, daß ein Fehler dabei die eingelegte Diskette beim Durchlaufen des Programms modifizieren und damit unbrauchbar machen kann. Verwenden Sie als Versuchskaninchen daher immer eine unwichtige Diskette, auf der keine einmalige Daten enthalten sind!

Die Programmierung dessen beginnt wieder mit ähnlichen Initialisierungen wie bei der Sprachausgabe. Hier erst einmal die Initialisierungs-Routinen für das Programm:

```
;**  direkter Diskettenzugriff über die Trackdisk.device **

OpenLib=-408
closelib=-414
ExecBase=4

Open=-30
Close=-36
opendevice =-444
CloseDev =-450
SendIo =-462
Read=-30-12
Write=-30-18
WaitForCh=-30-174
mode_old=1005

run:
     bsr     init            ;Initialisierung
     bra     test            ;System-Test
```

```
init:                       ;System initialisieren und öffnen
     move.l  execbase,a6        ;Zeiger auf EXEC-Bibliothek
     lea     dosname,a1
     moveq   #0,d0
     jsr     openlib(a6)        ;DOS-Library öffnen
     move.l  d0,dosbase
     beq     error

     lea     diskio,a1          ;Zeiger auf Disk-I/O-Bereich
     move.l  #diskrep,14(a1)    ;Zeiger auf Port
     clr.l   d0                 ;Laufwerk 0 (eingebaut)
     clr.l   d1                 ;keine Flags
     lea     trddevice,a0       ;Zeiger auf Device-Namen
     jsr     opendevice(a6)     ;Trackdisk.device öffnen
     tst.l   d0                 ;Fehler?
     bne     error              ;ja!

     move.l  #consolname(pc),d1 ;Consol-Definition
     move.l  #mode_old,d2        ;Modus: alt
     move.l  dosbase,a6          ;DOS-Basisadresse
     jsr     open(a6)            ;Fenster öffnen
     tst.l   d0                  ;Fehler?
     beq     error               ;ja!
     move.l  d0,conhandle        ;sonst Handle retten

     rts                         ;fertig

test:                           ;Platz für Test-Routinen
```

Und dann auch noch die nötigen Funktionen, die bei Beendigung des Programms für die diversen Abmeldungen sorgt:

```
error:
     move.l #-1,d7        ;Flag für Fehler (für SEKA)
qu:
     move.l  execbase,a6 ;EXEC-Basisadresse
     lea     diskio,a1   ;Zeiger auf Disk-I/O
     move.l  32(a1),d7   ;IO_ACTUAL in D7 (zum Testen)
     move    #9,28(a1)   ;Kommando: Motor ein/aus:
     move.l  #0,36(a1)   ;0=aus, 1=an, also Motor aus-
     jsr     sendio(a6)  ;schalten

     move.l  conhandle,d1 ;Fenster schließen
     move.l  dosbase,a6
     jsr     close(a6)
```

```
        move.l  dosbase,d1  ;DOS.Lib schließen
        move.l  execbase,a6
        jsr     closelib(a6)

        lea     diskio,a1
        jsr     closedev(a6) ;Trackdisk.device schließen

        rts
```

Und nicht zu vergessen die Routine, die auf einen Return-Tastendruck wartet, damit wir den Effekt der Testfunktion in Ruhe abwarten können:

```
getchr:                     ;ein Zeichen von der Tastatur holen
        move.l  #1,d3               ;1 Zeichen
        move.l  conhandle,d1        ;Fensterhandle
        move.l  #inbuff,d2          ;Buffer-Adresse
        move.l  dosbase,a6          ;DOS-Basisadresse
        jsr     read(a6)            ;Zeichen einlesen
        rts                         ;das war's
```

Zum Abschluß nun noch die Datenfelder und Texte, welche unser Programm braucht:

```
dosname:    dc.b  'dos.library',0
even
consolname: dc.b  'RAW:0/100/640/50/** Warte-Fenster',0
even
trddevice:  dc.b  'trackdisk.device',0
even
dosbase:    dc.l  0           ;DOS-Basisadresse
conhandle:  dc.l  0           ;Fenster-Handle
inbuff:     blk.b 80,0        ;Tastatur-Puffer
diskio:     blk.l 20,0        ;I/O-Struktur
diskrep:    blk.l 8,0         ;I/O-Port
diskbuff:   blk.b 512*2,0     ;Platz für 2 Sektoren
```

So, das war die Pflicht. Kommen wir nun zur Kür, und betrachten wir die Methode, Kommandos an die Diskettenstation zu übermitteln. Das erste Kommando, welches gleichzeitig das einfachste ist, tauchte schon auf: das Ein- und Ausschalten des

Laufwerksmotors. Die Nummer dieses Kommandos ist 9. Diese Nummer wird in das Kommandowort der I/O-Struktur eingetragen, welches im 28/29. Byte dieser Struktur liegt.

Ein Parameter muß nur noch zusätzlich übergeben werden, um zu bestimmen, ob der Motor an- oder ausgeschaltet werden soll. Diese Information kommt in das I/O-Längen-Langwort ab dem 36. Byte, und zwar eine Null für aus und eine Eins für Einschalten des Motors.

Welcher Motor dabei geschaltet werden soll, wird bereits beim Öffnen der Device mit angegeben. In D0 wird dabei nämlich die Nummer des gewählten Laufwerks übergeben, in unserem Fall ist dies die 0 für das eingabaute Laufwerk DF0.

Hier nun eine Übersicht über die Kommandos, mit denen wir auf die Diskette zugreifen können:

Nr.	Name	Funktion
2	*READ*	Lesen eines oder mehrerer Sektoren
3	*WRITE*	Schreiben von Sektoren
4	*UPDATE*	Zurückschreiben des Track-Puffers
5	*CLEAR*	Track-Puffer löschen
9	*MOTOR*	Motor ein- oder ausschalten
10	*SEEK*	Suchen eines Tracks
11	*FORMAT*	Formatieren von Tracks
12	*REMOVE*	Routine initialisieren, die beim Entfernen der Diskette aufgerufen wird
13	*CHANGENUM*	Anzahl der Diskwechsel feststellen
14	*CHANGESTATE*	Testen, ob Diskette eingelegt
15	*PROTSTATUS*	Testen, ob Diskette schreibgeschützt

Kommando Nr. 9 haben wir bereits kennengelernt. Schauen wir uns nun die 3 Kommandos an, mit welchen wir einige Abfragen machen können. Dies sind die letzten 3 Kommandos, die einen Rückgabewert in dem Langwort ab dem 32 Byte der I/O-Struktur ablegen. Dieser Wert wird in obigem Programm zum

Testen in das Register D7 geschrieben, dessen Inhalt Sie nach dem Durchlauf des Programms mit dem SEKA direkt ablesen können.

Hier die einfache Routine, mit der Sie eines dieser Kommandos ausführen lassen können:

```
test:
     lea     diskio,a1        ;Zeiger auf I/O-Struktur
     move    #13,28(a1)       ;Kommando übergeben (z.B. 13)
     move.l  execbase,a6      ;EXEC-Basisadresse in A6
     jsr     SendIO(a6)       ;Funktion aufrufen
```

Wird als Kommando CHANGENUM (13) übergeben, so erhält man in D7 die Zahl zurück, wie oft die bzw. eine Diskette in das Laufwerk rein und rausgetan wurde. Rufen Sie also das Programm auf, so erhalten Sie einen Wert zurück. Wenn Sie dann die Diskette herausnehmen und wieder einlegen, so wird diese Zahl beim erneuten Aufruf um 2 höher sein.

Das Kommando CHANGESTATE (14) ergibt die Information, ob eine Diskette eingelegt ist oder nicht. Ist eine Diskette im Laufwerk, so wird eine 0, andernfalls der Wert $FF zurückgegeben.

Die gleichen Wert erhält man auch von der PROTSTATUS-Funktion (a5) zurück. Hier bedeutet eine 0, daß die Diskette nicht schreibgeschützt ist, $FF wird bei einer geschützten Diskette übergeben.

Nun zu den Schreib-/Lese-Funktionen READ und WRITE. Diese Operationen benötigen natürlich einige Parameter mehr als die Statusfunktionen. Es müssen übergeben werden:

die Adresse des Ein-/Ausgabepuffers im Datenzeiger, die Anzahl der zu übertragenden Datebytes in I/O-Länge und die Datenadresse auf Diskette in I/O-Offset.

Die Anzahl der Datenbytes muß ein ganzzahliges Vielfaches von 512 sein, da jeder Sektor 512 Bytes lang ist und nur ganze Sektoren gelesen werden können.

Die Datenadresse bedeutet die Nummer des ersten Bytes auf der Diskette, ab dem die Daten übertragen werden sollen. Soll also der erste Sektor verwendet werden, so ist der Offset 0, beim 2. Sektor ist er 512 usw., also:

```
Offset=(Sektornummer-1)*512
```

Hier die Routine, mit der die ersten beiden Sektoren der Diskette in den Puffer geladen werden können:

```
test:
     lea diskio,a1
     move    #2,28(a1)        ;Command: READ
     move.l  #diskbuff,40(a1) ;Puffer
     move.l  #2*512,36(a1)    ;Länge:  2 Sektoren
     move.l  #0*512,44(a1)    ;Offset: 0 Sektoren
     move.l  execbase,a6      ;EXEC-Basisadresse
     jsr     SendIO(a6)       ;Funktion starten
```

Starten Sie einmal dieses Programm und sehen sich dann den Inhalt des Puffers mit 'q puffer' an. Hier können Sie nun erkennen, mit welchem Format die Disketten formatiert werden. Falls Sie belegte Sektoren lesen wollen, so ändern Sie doch die 0 in der Offset-Definition in 700 und starten erneut. Dort werden mit einiger Wahrscheinlichkeit irgendwelche Daten liegen.

Wenn Sie die Daten, welche Sie aus der Diskette gelesen haben, nun ändern und zurückschreiben wollen, so müssen Sie das WRITE-Kommando (3) verwenden. Die Parameter bleiben gleich. Doch Vorsicht: versuchen Sie nicht, mit der 'm'-Funktion des SEKA die Daten zu ändern und dann das Programm mit dem geänderten Disk-Kommando neu zu assemblieren! Dabei wird der Puffer nämlich wieder mit Nullen gefüllt und alle Daten sind weg!

Wenn Sie nun dennoch das WRITE-Kommando übergeben haben, so wundern Sie sich bestimmt, warum das Diskettenlaufwerk aus bleibt. Dies liegt einfach daran, daß der Amiga einen gelesenen Track in einen eigenen Puffer schreibt und beim

WRITE-Kommando die neuen Daten lediglich dort hineinschreibt. Erst wenn auf einen anderen Track zugegriffen werden soll, werden diese Daten wieder auf die Diskette geschrieben.

Sie können dieses Zurückschreiben der Daten jedoch auch direkt ausführen lassen. Hierzu dient das Kommando UPDATE (4).

Ein weiteres interessantes Kommando ist FORMAT (11). Dieses Kommando benötigt ein Datenfeld, welches 11*512=5632 Bytes lang ist, also so lang wie der ganze Track Daten aufnehmen kann. Außerdem muß die Angabe des Offset ein Vielfaches dieser Zahl sein, damit auch am Anfang eines Tracks begonnen wird.

Die Angabe der Länge muß schließlich ebenfalls ein Vielfaches von 5632 sein. Sollen mehrere Tracks formatiert werden, so wird jeder Track mit den selben Daten gefüllt.

Mit dieser Funktion kann also leicht ein Disketten-Kopierprogramm geschrieben werden, welches erst den Inhalt der Quelldiskette mit READ teilweise in den Speicher lädt und dann nach Einlegen der Zieldiskette für jeden Track einen Format-Befehl ausführt, bei dem jeweils die Adresse der entsprechenden Daten im Speicher angegeben wird. So funktioniert auch das DiskCopy-Programm, welches ja grundsätzlich die Zieldiskette neu formatiert.

Das Kommando SEEK (10) benötigt lediglich die Angabe des Offsets. Es fährt dann den Schreib-/Lesekopf des Laufwerks in die angegeben Position, ohne jedoch irgendeinen Zugriff auf die Diskette oder einen Test auf korrekte Position auszuführen.

Das REMOVE-Kommando (12) schließlich dient zum Installieren einer Interrupt-Routine, welche beim Entfernen der Diskette aus dem Laufwerk aufgerufen wird. Die Adresse der Interrupt-Struktur wird im Datenzeiger der I/O-Struktur übergeben. Wird hier eine Null eingetragen, so wird die Interrupt-Routine abgeschaltet.

7. Die Arbeit mit Intuition

Nachdem wir so viel in unserem Rechner mit der Maschinensprache anfangen können, wollen wir nun auf die Besonderheiten des Amiga eingehen. Gemeint ist hier das Betriebssystem Intuition, welches die Verwaltung von Fenstern, Bildschirmen (Screens), der Maus und vielem mehr erledigt. Bevor wir uns an die Programmierung dieser Leckerbissen des Amiga machen, habe ich noch eine gute und eine schlechte Mitteilung für Sie:

Erst die gute: das Intuition besitzt so viele Funktionen und Möglichkeiten, daß man ungeheuer flexibel in der Programmierung eigener Ideen ist. Doch nun auch den Nachteil: um diese hohe Flexibilität zu gewährleisten, müssen eine Menge Parameter vorbereitet und übergeben werden, was eine gewisse Fleißarbeit erfordert.

Aber dies ist kein Grund zur Panik: wenn man sich einmal die notwendigen Gerüste gebaut, sprich programmiert, hat, wird die Programmierung und vor allem das Ausprobieren immer interessanter. Allerdings sollten Sie jeweils vor dem Ausprobieren neuer Variationen immer Ihr Programm abspeichern, da das Intuition auf falsche Parameter manchmal empfindlich reagiert und dann über Ihre vermutliche Absicht meditiert...

Doch nun ran ans Werk! Um die Arbeit mit Intuition zu beginnen, brauchen wir natürlich auch die Intuition-Bibliothek, welche wir ebenso wie z.B. die DOS-Bibliothek mittels der EXEC-Funktion OpenLibrary laden. Hier der entsprechende Programmteil, welcher wohl nicht mehr besonders besprochen werden muß:

```
OpenLib   = -408
ExecBase  =  4

run:
     bsr     openint          ;Intuition-Bibliothek laden
     ...
```

```
openint:        ;* System initialisieren und öffnen
     move.l  ExecBase,a6       ;EXEC-Basisadresse
     lea     IntName,a1        ;Name der Intuition-Bibliothek
     jsr     OpenLib(a6)       ;Intuition öffnen
     move.l  d0,intbase        ;Intuition-Basisadresse retten
     rts

IntName: dc.b "intuition.library",0
even
intbase: dc.l 0                ;Basisadresse des Intuition
```

Wenn unser Programm nachher fertig ist, muß nach Abschluß der Arbeit und dem eventuellen Schließen der Screens und Fenster auch diese Bibliothek wieder geschlossen werden. Dafür nimmt man die ebenfalls bekannte CloseLibrary-Funktion des EXEC mit dem Offset -414. Hier die entsprechende Unterroutine:

```
CloseLibrary  = -414

     ...
closeint:                      ;* Intuition schließen
     move.l  execbase,a6       ;EXEC-Basisadresse in A6
     move.l  intbase,a1        ;Intuition-Basisadresse in A1
     jsr     CloseLibrary(a6)  ;Intuition schließen
     rts                       ;fertig
```

Wenn dies alles auch geklappt hat, können wir nun endlich den ersten Schritt in der Arbeit mit Intuition wagen.

7.1 Bildschirm öffnen

Das Intuition ist ein graphisches Betriebssystem. Aus diesem Grund ist es dafür angelegt, mit dem Bildschirm zu operieren. Interessanterweise ist es sogar in der Lage, mit mehreren Bildschirmen gleichzeitig zu arbeiten, obwohl man nur einen Monitor an den Amiga anschließen kann!

Des Rätsels Lösung sind die Screens, was zu deutsch auch nichts anderes als Bildschirme heißt. Man kann sich beliebig viele (na ja, wenigstens einige, solange der Speicher noch mitmacht) sol-

che Screens öffnen, auf denen sich dann jeweils Fenster öffnen, Menüs darstellen und Ein-/Ausgaben machen lassen. Die einzelnen Screens sind voneinander völlig unabhängig und lassen sich alle gleichzeitig auf dem einen Monitor bearbeiten. Der Haken an der Sache ist jedoch, daß man nur sehen kann, was 'vorne' auf dem Bildschirm liegt.

Sie können die einzelnen Screens nach Belieben nach vorne oder hinten bringen bzw. hoch- und runterschieben. Außerdem kann durch Betätigung der linken Amiga-Taste und 'm' der Screen der Workbench nach vorne bringen, wenn man sich in dem 'Stapel' der Screens verlaufen hat.

Wir wollen nun mit der Programmierung des Intuition beginnen, indem wir uns einen eigenen Screen anlegen. Die Intuition-Bibliothek haben wir geladen und brauchen jetzt eigentlich nur die Funktion OpenScreen aufzurufen.

Doch Moment einmal! Wie soll denn der Screen aussehen, wohin soll er und welchen Aufbau soll er haben? Wir müssen uns zuerst einmal ansehen, welche Möglichkeiten wir bei der Gestaltung unseres Screens überhaupt haben.

Die Angaben über den zu öffnenden Screen werden in Form einer Tabelle abgelegt, welche 13 Einträge besitzt. Betrachten wir nun all diese einzelnen Parameter, welche unseren Screen bestimmen.

Beginnen wir die Tabelle mit dem Label 'screen_defs', welcher an einer geraden Adresse stehen muß:

```
even
screen_defs:        ;* hier beginnt die Screen-Tabelle
```

Die ersten Angaben, die wir über den Screen machen können, sind die Position und Größe. Sagen wir also, er soll ganz links oben beginnen und den vollen Bildschirm einnehmen. Dadurch erhalten wir die Position X=0 und Y=0, die Breite 320 und die Höhe 200. Diese Angaben sind in Bildschirmpunkten gemeint, so daß unser Screen die volle Größe bekommt.

```
x_pos:    dc.w     0    ;X-Position
y_pos:    dc.w     0    ;Y-Position
width:    dc.w   320    ;Breite
heigth:   dc.w   200    ;Höhe
```

Als nächstes folgt die Überlegung, wieviele Farben darstellbar sein sollen. Dies wird durch die Anzahl der Bitplanes, auch Tiefe genannt, bestimmt. Nehmen wir dafür eine 2, so können insgesamt 2^2, also 4 Farben dargestellt werden, bei einer 1 nur zwei Farben. Wählen wir zunächst die 2, da 4 Farben üblicherweise ausreichen.

```
depth:    dc.w     2    ;Anzahl der Bitplanes
```

Nun bestimmen wir die Farben, in denen Titelzeile und Funktionssymbole gezeichnet werden soll, indem wir die Nummer des Farbregisters dafür angeben:

```
detail_pen:  dc.b   0    ;Farbe des Textes ect.
```

Und nun auch die Farbe des Text-Hintergrundes:

```
block_pen:   dc.b   1    ;Hintergrund-Farbe
```

Beachten Sie bitte, daß diese beiden Angabe in Bytegröße vorliegen müssen! Die Farben sind, wenn sie nicht geändert wurden, normalerweise folgende (beachten Sie bitte, daß die Anzahl der Farben von der Anzahl der Bitmaps abhängt!):

Pen	**Farbe**
0	Hintergrund (blau)
1	weiß

bei 2 Bitplanes:

2	schwarz
3	rot

bei 3 Bitplanes:

4	blau
5	violett
6	türkis
7	weiß

bei 4 Bitplanes:

8	schwarz
9	rot
0	grün
11	braun
12	blau
13	blau
14	grün
15	grau

Nun folgt ein Wort, in dem die einzelnen Bits die Erscheinungsform des Screens beschreiben. Diese Bits sind:

Bit	Wert	Name	Bedeutung
1	2	GENLOCK_VIDEO	
2	4	INTERLACE	Schaltet den Screen in den Interlace-Betrieb. Die Auflösung und damit die maximalen Werte der Screengröße verdoppeln sich somit.
6	$40	PFBA	
7	$80	EXTRA_HALFBRITE	
8	$100	GENLOCK_AUDIO	
10	$400	DBLPF	Bewirkt die Aufspaltung des Screens in Rahmen- und Zeichenbereich
11	$800	HOLDNMODIFY	Schaltet den Hold-and-Modify-Modus ein
13	$2000	VP_HIDE	
14	$4000	SPRITES	Ermöglicht die Verwendung von Sprites
15	$8000	MODE_640	Schaltet den Screen in die hochauflösende Graphik (640x400)

Wir wählen für unseren Beispiel-Screen den Wert 2 (normal):

```
view_modes:  dc.w   2    ;Darstellungs-Modus
```

Das nun folgende Wort ist wieder so aufgebaut, daß manches Bit eine eigene Bedeutung hat. Mit diesem Wort wird gewählt, welcher Art der Screen sein soll. Setzen wir hier eine 15 ein, so wird unser Screen ein 'Customscreen', was uns alle Möglichkeiten bietet.

```
screen_type: dc.w  15    ;Screen-Typ: Customscreen
```

Nun folgt ein Zeiger auf den zu verwendende Zeichensatz für alle Ausgaben in diesem Screen. Will man hier keinen eigenen Zeichensatz installieren, so reicht es, wenn man hier eine Null einsetzt: es wird dann der Standard-Zeichensatz verwendet.

```
font:      dc.l      0    ;Zeichensatz: Standard
```

Nun noch ein Zeiger, der auf den Text zeigt, welcher die Überschrift des Screens bilden soll. Dieser Text muß mit einer 0 abgeschlossen sein, wie man es bereits von Fenster-überschriften her kennt.

```
title:     dc.l   titel   ;Zeiger auf Titel-Text
```

Als nächstes folgt ein Langwort, mit dem die sogenannten Gadgets definiert werden. Diese Gadgets stellen die Funktionssymbole dar, welche z.B. zum 'Nachvornebringen' des Screens angeklickt werden können. Das in dieser Tabelle einzutragende Langwort stellt einen Zeiger auf eine Liste dar, in denen die Gadgets aufgezählt werden. Dabei sind allerdings nicht die System-Gadgets mit aufgeführt. Wir benutzen hier aber nur diese System-Gadgets, so daß wir hier eine Null eintragen.

```
gadgets:  dc.l      0    ;keine Gadgets
```

Schließlich und endlich kommt nun noch ein Langwort, welches wir aber nur benötigen, wenn wir für unseren Screen eine eigene Bitmap verwenden wollen. Da dies nicht der Fall ist, schreiben wir auch hier eine Null.

```
bitmap:   dc.l      0    ;keine Bitmap
```

Dies waren alle Listeneinträge, die wir zur Definition unseres Screens benötigen. Was noch fehlt, ist natürlich der Text der Überschrift, was nach unserem Beispiel etwa so programmiert wird:

```
titel:    dc.b  'Unser Screen',0   ;Screen-Überschrift
```

Hier noch einmal die ganze Liste im Überblick:

```
even
screen_defs:          ;* hier beginnt die Screen-Tabelle
x_pos:        dc.w      0    ;X-Position
y_pos:        dc.w      0    ;Y-Position
width:        dc.w    320    ;Breite
heigth:       dc.w    200    ;Höhe
depth:        dc.w      2    ;Anzahl der Bitplanes
detail_pen:   dc.b      0    ;Farbe des Textes ect.
block_pen:    dc.b      1    ;Hintergrund-Farbe
view_modes:   dc.w      2    ;Darstellungs-Modus
screen_type:  dc.w     15    ;Screen-Typ: Customscreen
font:         dc.l      0    ;Zeichensatz: Standard
title:        dc.l  titel    ;Zeiger auf Titel-Text
gadgets:      dc.l      0    ;keine Gadgets
bitmap:       dc.l      0    ;keine Bitmap

titel:        dc.b  'Unser Screen',0   ;Screen-Überschrift
```

Mit den so vorbereiteten Parametern ist es nun sehr einfach, den Screen zu öffnen. Wir benötigen dafür die OpenScreen-Funktion des Intuition, welche den Offset -198 besitzt und lediglich einen Parameter übergeben bekommt, nämlich die Adresse der oben aufgezeigten Parameter-Tabelle. Der entsprechende Programmteil lautet daher:

```
OpenScreen = -198

     bsr     openint        ;Intuition öffnen
     bsr     scropen        ;Screen öffnen
     ...
```

```
scropen:                    ;* Screen öffnen
     move.l  intbase,a6     ;Intuition-Basisadresse in A6
     lea     screen_defs,a0 ;Zeiger auf Tabelle
     jsr     openscreen(a6) ;und öffnen
     move.l  d0,screenhd    ;Handle des Screens retten
     rts                    ;Rückkehr ins Hauptprogramm

     ...

screen_defs:                ;Tabelle s. o.
```

Und schon ist der Bildschirm des Amiga von unserem eigenen Screen überdeckt. Nun können mit ihm alle möglichen Dinge angestellt werden, bis das Programm seine Arbeit beendet hat. Danach muß der Screen aber auch wieder geschlossen werden, damit man wieder den Workbench-Screen sehen kann.

Zum Schließen des Screens bedienen wir uns der CloseScreen-Funktion mit dem Offset -66, der wir nur den aus der OpenScreen-Funktion erhaltenen Zeiger auf die Screen-Struktur übergeben brauchen:

```
CloseScreen  = -66

     ...
scrclose:                     ;* Screen schließen
     move.l  intbase,a6       ;Intuition-Basisadresse in A6
     move.l  screenhd,a0      ;Screen-Handle in A0
     jsr     CloseScreen(a6)  ;Screen schließen
     rts                      ;fertig
```

Betrachten wir nun das Langwort, welchen uns die OpenScreen-Funktion zurückgegeben hat. Dies ist ein Zeiger und zeigt auf eine Screen-Struktur, in der alle nötigen Daten über den Screen enthalten sind. Außer den von uns vorgegebenen Daten sind da noch Zeiger auf die Bildschirmbereiche für die einzelnen Bit-Planes usw.

Der Aufbau dieser Struktur ist recht kompliziert und enthält einige Daten, die man nicht gebrauchen kann. Einige der Einträge jedoch sind durchaus interessant. Hier eine Auswahl von verwendbaren Parametern:

Nr.	Name	Funktion
0	*(NextScreen.L)*	Zeiger auf den nächsten Screen
4	*(FirstWindow)*	Zeiger auf die erste Fenster-Struktur
8	*(LeftEdge.W)*	
$A	*(TopEdge.W)*	Position des Screens
$C	*(Width.W)*	Breite
$E	*(Height.W)*	Höhe
$10	*(MouseY.W)*	
$12	*(MouseX.W)*	Mausposition im Screen
$14	*(Flags.W)*	Screen-Flags
$16	*(Title.L)*	Zeiger auf Titeltext
$1A	*(DefaultTitle)*	Zeiger auf Normal-Titel
$28	*(Font.L)*	Zeiger auf Zeichensatz
$C0	*(Plane0.L)*	Zeiger auf die Bit-Plane 0
$C4	*(Plane1.L)*	Zeiger auf die Bit-Plane 1
$C8	*(Plane2.L)*	Zeiger auf die Bit-Plane 2
$CC	*(Plane3.L)*	Zeiger auf die Bit-Plane 3

Ein Beispiel für die Verwendung der Plane-Zeiger wäre z.B., wenn Sie eigene Zeichenroutinen geschrieben haben und anwenden wollen. Sie laden dann nach dem Öffnen des Screens mit den folgenden Anweisungen die Adresse einer Plane in ein Adreßregister:

```
        move.l  screenhd,a5     ;Screen-Zeiger in A5
        move.l  $c0(a5),a5      ;Bit-Plane 0-Zeiger in A5
```

Wenn Sie dies ausprobieren wollen, können Sie ja mal vor dem 'loop'-Label des Programms schreiben:

```
        move.l  screenhd,a5     ;Screen-Zeiger in A5
        move.l  $c0(a5),a5      ;Bit-Plane 0-Zeiger in A5
        move    #$20,d0         ;Zähler D0=$20
lop1:
        move    d0,(a5)         ;Zählerbits ins Bild schreiben
        add.l   #80,a5          ;Adresse +80, nächste Zeile
        dbra    d0,lop1         ;weiter, bis D0<0
```

Dieser Programmteil zeichnet untereinander in die linke obere Ecke des Screens ein senkrechtes weißes Muster, welches dem Bitmuster der Zahlen $20 bis 0 entspricht. Dies ist zugegebenermaßen kein besonders sinnvolles Programm, aber es verdeutlicht, wie einfach man aus einem Maschinenprogramm heraus direkt ins Bild schreiben kann. Wenn Sie nun den Offset in der zweiten Zeile in $C4 ändern, wird das Muster in rot ausgegeben.

Sie können den gesamten Screen mit der bekannten Technik verschieben, indem Sie den Mauszeiger auf den oberen Rand bringen und mit gedrückter linken Maustaste hoch- und runterschieben. Doch was der Benutzer mit der Maus kann, können wir mit einem Programm schon lange: Diese Verschiebung ist natürlich auch vom Programm aus möglich.

Betrachten wir einmal diese Verschiebung des Screens ohne die Maus. Nehmen wir dafür zur Demonstration den Joystick, welchen wir in Port 2 einstecken. Wie im Kapitel über die Hardware-Register bereits erklärt, kann die Stellung dieses Joysticks aus der Speicherzelle $DFF00C ausgelesen werden. Hier erfahren wir also für unser Beispielprogramm, in welche Richtung der Screen bewegt werden soll.

Das Verschieben wird natürlich durch eine weitere Funktion des Intuition bewerkstelligt. Es handelt sich dabei um die Funktion MoveScreen mit dem Offset -162, die drei Parameter benötigt:

In A0 den Zeiger auf die Screen-Struktur, welchen wir nach dem Öffnen des Screens in D0 erhalten und in 'screenhd' gerettet haben.

In D1 die gewünschte Verschiebung in Y-Richtung, also vertikal.

In D0 die horizontale Verschiebung in X-Richtung. Diese Variante funktioniert leider nicht, so daß wir den Screen nur vertikal verschieben können.

Wir fügen also in unser Programm folgende Zeilen ein:

```
MoveScreen   = -162

     ...

scrmove:                        ;*  Screen um D0  nach  rechts
                                 und um  D1 nach unten schieben
     move.l  intbase,a6         ;Intuition-Basisadresse in A6
     move.l  screenhd,a0        ;Screen-Handle in A0
     clr.l   d0                 ;keine horizontale Verschiebung
     jsr     MoveScreen(a6)     ;und verschieben
     rts                        ;fertig
```

Und nun schreiben wir das erste vollständige Programm, welches folgende Schritte abläuft:

1. Öffnen der Intuition-Bibliothek
2. Öffnen eines Screens
3. Verschieben des Screens in der durch den Joystick in Port 2 vorgegebenen Richtung
4. bei Betätigung des Feuerknopfes Schließen des Screens
5. Schließen der Intuition-Bibliothek
6. Ende

Für das Ende verwenden wir ein RTS, wodurch wir das Programm auch vom SEKA aus mit 'j run' starten können.

Und hier das komplette Programm, inclusive der Unterroutinen, damit Sie diese auf einen Blick überschauen können:

```
;**  Demo-Programm zum Öffnen und Verschieben eines  Screens **

MoveScreen   = -162
OpenScreen   = -198
CloseScreen  = -66
CloseLibrary = -414
OpenLib      = -408            ;Library öffnen
ExecBase     =  4              ;EXEC-Basisadresse
```

```
joy2        = $dff00c       ;Joystick 2 Daten
feuer       = $bfe001       ;Feuerknopf 2: Bit 7

run:
     bsr    openint         ;Intuition öffnen
     bsr    scropen         ;Screen öffnen

     move   joy2,d6         ;Joystick-Grunddaten retten

loop:
     tst.b  feuer           ;Feuerknopf testen
     bpl    ende            ;gedrückt: Ende

     move   joy2,d0         ;Grunddaten in D0
     sub    d6,d0           ;neue Daten abziehen

     cmp    #$0100,d0       ;hoch ?
     bne    noup            ;nein
     move.l #-1,d1          ;dy=-1
     bsr    scrmove         ;hochschieben
     bra    loop

noup:
     cmp    #$0001,d0       ;runter ?
     bne    loop            ;nein
     move.l #1,d1           ;dy=1
     bsr    scrmove         ;runterschieben
     bra    loop

ende:
     bsr    scrclose        ;Screen schließen
     bsr    closeint        ;Intuition schließen
     rts                    ;fertig !


openint:       ;* System initialisieren und öffnen
     move.l ExecBase,a6     ;EXEC-Basisadresse
     lea    IntName,a1      ;Name der Intuition-Bibliothek
     jsr    OpenLib(a6)     ;Intuition öffnen
     move.l d0,intbase      ;Intuition-Basisadresse retten
     rts
```

```
closeint:                       ;* Intuition schließen
     move.l  execbase,a6        ;EXEC-Basisadresse in A6
     move.l  intbase,a1         ;Intuition-Basisadresse in A1
     jsr     CloseLibrary(a6)   ;Intuition schließen
     rts                        ;fertig

scropen:                        ;* Screen öffnen
     move.l  intbase,a6         ;Intuition-Basisadresse in A6
     lea     screen_defs,a0     ;Zeiger auf Tabelle
     jsr     openscreen(a6)     ;und öffnen
     move.l  d0,screenhd        ;Handle des Screens retten
     rts                        ;Rückkehr ins Hauptprogramm

scrclose:                       ;* Screen schließen
     move.l  intbase,a6         ;Intuition-Basisadresse in A6
     move.l  screenhd,a0        ;Screen-Handle in A0
     jsr     CloseScreen(a6)    ;Screen schließen
     rts                        ;fertig

scrmove:                        ;*  Screen um D0 nach rechts und um
D1
                                 nach unten schieben
     move.l  intbase,a6         ;Intuition-Basisadresse in A6
     move.l  screenhd,a0        ;Screen-Handle in A0
     clr.l   d0                 ;keine horizontale Verschiebung
     jsr     MoveScreen(a6)     ;und verschieben
     rts                        ;fertig

even
screen_defs:        ;* hier beginnt die Screen-Tabelle
x_pos:       dc.w      0    ;X-Position
y_pos:       dc.w      0    ;Y-Position
width:       dc.w    320    ;Breite
heigth:      dc.w    200    ;Höhe
depth:       dc.w      2    ;Anzahl der Bitplanes
detail_pen:  dc.b      1    ;Farbe des Textes = weiß
block_pen:   dc.b      3    ;Hintergrund-Farbe = rot
view_modes:  dc.w      2    ;Darstellungs-Modus
screen_type: dc.w     15    ;Screen-Typ: Customscreen
font:        dc.l      0    ;Zeichensatz: Standard
title:       dc.l  titel    ;Zeiger auf Titel-Text
gadgets:     dc.l      0    ;keine Gadgets
bitmap:      dc.l      0    ;keine Bitmap
```

```
intbase:    dc.l      0    ;Basisadresse des Intuition
screenhd:   dc.l      0    ;Screen-Handle
IntName:    dc.b  'intuition.library',0
even
titel:      dc.b  'Unser Screen',0   ;Screen-Überschrift
even
```

Sie sehen an Hand dieses Beispiels, wie einfach sich das sogenannte Scrolling hier realisieren läßt, was gerade bei Spielen sehr beliebt ist.

Eine weitere nette Spielerei läßt sich durch die Funktion DisplayBeep erreichen, welche den Offset -96 hat und als Parameter nur den Screen-Zeiger aus 'screenhd' in A0 übergeben bekommt. Diese Funktion läßt den Screen kurz in oranger Farbe aufblitzen. Ein eventuell sichtbarer anderer Screen bleibt davon unberührt. Die Beep-Funktion wird folgendermaßen ausgelöst:

```
DisplayBeep  = -96

     ...
     move.l  intbase,a6        ;Intuition-Basisadresse in A6
     move.l  screenhd,a0       ;Screen-Zeiger in A0
     jsr     DisplayBeep(a6)   ;Screen aufblitzen lassen
```

Wenn Sie in A0 anstelle des Screenzeigers eine Null übergeben, so blinkt der gesamte Bildschirm auf.

Schön und gut, jetzt haben wir einen eigenen Screen, den wir sogar hoch- und runterschieben können. Doch was nützt uns dies, wenn wir darauf nichts darstellen können? Lassen Sie uns daher nun ein Fenster auf diesem Screen öffnen!

7.2 Fenster öffnen

Wie Sie bereits im Kapitel über Programmvorbereitung gesehen haben, ist es recht einfach, mittels der DOS-Bibliothek ein Fenster zu öffnen. Diese Methode versagt aber leider, wenn wir in einem eigenen Screen arbeiten wollen. Wir müssen daher eine andere Methode kennenlernen, mit der beliebige Fenster auf beliebigen Screens geöffnet werden können.

Auch hierfür bietet das Intuition natürlich eine Funktion, die sich bezeichnenderweise OpenWindow nennt. Sie trägt den Offset -204 und benötigt nur einen Parameter, und zwar, ähnlich der OpenScreen-Funktion, einen Zeiger auf eine Fenster-Definitions-Tabelle. Dieser Zeiger wird im Register A0 übergeben.

Diese Tabelle ist derjenigen der Screen-Definition sehr ähnlich. Auch hier werden zuerst 4 Worte eingetragen, in denen X- und Y-Position sowie Breite und Höhe des zu öffnenden Fensters vorgegeben werden. Dazu ein Beispiel:

```
even
window_defs:
    dc.w    10      ;X-Position
    dc.w    20      ;Y-Position
    dc.w   300      ;Breite
    dc.w   150      ;Höhe
```

Danach folgen die beiden Bytes, welche die Schrift- und Hintergrundfarbe definieren:

```
    dc.b     1      ;Schriftfarbe weiß
    dc.b     3      ;auf rotem Hintergrund
```

Das nun folgende Langwort enthält in seinen Bits die sogenannten IDCMP-Flags, mit denen bestimmt wird, bei welchem Ereignis das Intuition dem Programm eine Meldung übermitteln soll. Die einzelnen Bits haben folgende Bedeutung (Wert und Name des Bits in Klammern):

Bit	Wert	Name	Bedeutung: Nachricht bei
0	$000001	SIZEVERIFY	
1	$000002	NEWSIZE	Veränderung der Fenstergröße
2	$000004	REFRESHWINDOW	
3	$000008	MOUSEBUTTONS	Drücken einer Maustaste
4	$000010	MOUSEMOVE	Bewegungen der Maus
5	$000020	GADGETDOWN	Auswahl eines speziellen Gadgets
6	$000040	GADGETUP	wie oben
7	$000080	REQSET	
8	$000100	MENUPICK	Auswahl eines Menüpunktes
9	$000200	CLOSEWINDOW	Schließen des Fensters
10	$000400	RAWKEY	Drücken einer Taste
11	$000800	REQVERIFY	
12	$001000	REQCLEAR	
13	$002000	MENUVERIFY	
14	$004000	NEWPREFS	Ändern der Preferences
15	$008000	DISKINSERTED	Einlegen einer Diskette
16	$010000	DISKREMOVED	Herausnehmen einer Diskette
17	$020000	WBENCHMESSAGE	
18	$040000	ACTIVEWINDOW	Aktivieren des Fensters
19	$080000	INACTIVEWINDOW	Desaktivieren des Fensters
20	$100000	DELTAMOVE	Mausbewegungen relativ melden

Wir werden unser erstes Fenster vorläufig nur auf Betätigung des Schließ-Symbols reagieren lassen, so daß wir schreiben:

```
dc.l    $200    ;IDCMP-Flags: CLOSEWINDOW
```

Nun kommt noch ein Langwort, in welchem die Bits den Typ des Fensters bestimmen. Hiermit kann also das Fenster wirklich beliebig aufgebaut werden, im Gegensatz zu den mit der DOS-Funktion geöffneten Fenstern. Die Bits bedeuten:

Bit	Wert	Name	Bedeutung: Nachricht bei
0	$0000001	WINDOWSIZING	Fenstergröße veränderbar
1	$0000002	WINDOWDRAG	Fenster verschiebbar
2	$0000004	WINDOWDEPTH	Fensterüberlagerung möglich
3	$0000008	WINDOWCLOSE	Fensterschließ-Symbol
4	$0000010	SIZEBRIGHT	
5	$0000020	SIZEBBOTTOM	
6	$0000040	SIMPLE_REFRESH	Neuzeichnen manuell
7	$0000080	SUPER_BITMAP	ganzen Fensterinhalt speichern

8	$0000100	BACKDROP	Fenster nach hinten
9	$0000200	REPORTMOUSE	Mauskoordinaten melden
10	$0000400	GIMMEZEROZERO	Trennung von Leisten und Fenster
11	$0000800	BORDERLESS	Fenster ohne Ränder
12	$0001000	ACTIVATE	Fenster aktiv
13	$0002000	WINDOWACTIVE	
14	$0004000	INREQUEST	
15	$0008000	MENUSTATE	
16	$0010000	RMBTRAP	rechte Maustaste: kein Menü
17	$0020000	NOCAREREFRESH	keine Erneuerungsmeldung
24	$1000000	WINDOWREFRESH	
25	$2000000	WBENCHWINDOW	

Refresh heißt hier nichts anderes, als daß der Fensterinhalt evtl. neu aufgebaut werden muß, z.B. nach einer Vergrößerung des Fensters. Wird keiner der Refresh-Bits gesetzt, so bedeutet dies den Smart-Refresh-Modus, bei dem das Intuition selbst für die Erneuerung des Fensters sorgt. Dies ist auch die einfachste und somit übliche Methode.

Wenn wir also für unser Beispiel-Fenster als Typ den Wert $100F wählen, so wird das Fenster beim Öffnen sofort aktiv und trägt alle System-Gadgets:

```
        dc.l    $100f       ;ACTIVATE und alle Gadgets
```

Das nächste Langwort in der Liste ist dafür gedacht, daß Sie eigene Gadgets in das Fenster bringen wollen. Dann kommt hier nämlich ein Zeiger auf die Struktur diese Gadgets hin. Da wir dies nicht wollen, setzen wir hier eine Null ein.

```
        dc.l    0           ;FirstGadget: keine eigenen Gadgets
```

Auch das nächste Langwort, welches als Zeiger auf eine Graphik-Struktur für eigene Symbole zum Abhaken von Menüpunkten vorgesehen ist, setzen wir auf Null, um so die Standard-Zeichen zu setzen:

```
        dc.l    0           ;CheckMark: Standard
```

Der nächste Listeneintrag ist ein Zeiger auf den Text, welcher als Fenster-Überschrift dienen soll. Auch dieser Text muß mit einem Null-Byte abgeschlossen werden.

```
        dc.l    windowname  ;Zeiger auf Fenster-Titel
```

Nun folgt als Langwort der Zeiger auf die Screen-Struktur, welchen wir aus dem OpenScreen-Aufruf erhalten haben. Die einfachste Methode, dies hier einzutragen, ist die, an dieser Stelle den Puffer zum Retten dieses Zeigers einzusetzen:

```
screenhd:  dc.l   0       ;Screen-Zeiger
```

Das nächste Langwort dient als Zeiger auf eine Bitmap, wenn man für das Fenster eine eigene anlegen will. Da wir dies auch nicht wollen, setzen wir auch hier eine Null ein:

```
        dc.l      0       ;keine eigene Bitmap
```

Nun folgen vier Worte, in denen die Mindestbreite und -höhe sowie die maximale Breite und Höhe des Fensters vorgegeben werden:

```
        dc.w    150       ;Mindestbreite
        dc.w     50       ;Mindesthöhe
        dc.w    320       ;maximale Breite
        dc.w    200       ;maximale Höhe
```

Endlich das letzte Wort der Liste gibt den Screen-Typ an, in dem das Fenster liegen soll. Wir tragen hier wieder eine 15 ein, damit der Screen als Custom-Screen angegeben wird:

```
        dc.w     15       ;Screen-Typ: Custom-Screen
```

Hier noch einmal die ganze Liste im Überblick:

```
even
window_defs:
     dc.w    10          ;X-Position
     dc.w    20          ;Y-Position
     dc.w   300          ;Breite
     dc.w   150          ;Höhe
     dc.b     1          ;Schriftfarbe weiß
     dc.b     3          ;auf rotem Hintergrund
     dc.l  $200          ;IDCMP-Flags: CLOSEWINDOW
     dc.l $100f          ;ACTIVATE und alle Gadgets
     dc.l     0          ;FirstGadget: keine eigenen Gadgets
     dc.l     0          ;CheckMark: Standard
     dc.l windowname     ;Zeiger auf Fenster-Titel
screenhd:  dc.l   0      ;Screen-Zeiger
     dc.l     0          ;keine eigene Bitmap
     dc.w   150          ;Mindestbreite
     dc.w    50          ;Mindesthöhe
     dc.w   320          ;maximale Breite
     dc.w   200          ;maximale Höhe
     dc.w    15          ;Screen-Typ: Custom-Screen
```

Und natürlich noch der Fenster-Titel:

```
windowname:  dc.b  'Unser Fenster',0
even
```

Fügen Sie diese Zeilen einmal in das oben aufgelistete Programm ein. Dazu kommen außerdem noch die zwei Unterroutinen zum Öffnen und Schließen des Fensters:

```
OpenWindow   = -204
CloseWindow  = -72

     ...
windopen:
     move.l  intbase,a6        ;Intuition-Basis in A6
     lea     windowdef,a0      ;Zeiger auf Fenster-Definition
     jsr     openwindow(a6)    ;Fenster öffnen
     move.l  d0,windowhd       ;Fenster-Handle retten
     rts
```

```
windclose:
     move.l  intbase,a6        ;Intuition-Basis in A6
     move.l  windowhd,a0       ;Fenster-Handle
     jsr     closewindow(a6)   ;Fenster schließen
     rts

     ...
windowhd:    dc.l    0         ;Fenster-Handle
```

Sie können nun auch in dem Beispielprogramm hinter dem 'bsr scropen' ein 'bsr windopen' und vor dem 'bsr scrclose' ein 'bsr windclose' einfügen. Wenn Sie das Programm dann starten, verschieben Sie mit dem Screen auch das neue Fenster, da dies ja innerhalb des Screens liegt. Wenn Sie versuchen, das Fenster mit der Maus aus dem Screen hinauszuschieben, werden Sie feststellen, daß dies nicht möglich ist.

Das Fenster trägt in diesem Beispiel das Schließ-Gadget in der linken oberen Ecke, womit üblicherweise durch Anklicken ein Fenster geschlossen werden kann. Klicken Sie dies nun einmal an. Der Effekt: keiner.

Die Auswertung dieses und aller anderen Gadgets sowie anderer Ereignisse muß also ebenfalls programmiert werden, da das Intuition nicht weiß, welche Aktion auf welches Ereignis folgen soll. Betrachten wir daher im nächsten Kapitel diese Ereignis-Behandlung.

7.3 Meldungen und Abfragen: Requester

Sie kennen bestimmt die beliebte Meldung des Amiga "Please insert xxx in unit 0", wenn Sie nur ein Laufwerk besitzen. Diese Meldung wird in einem Extrafenster ausgegeben, welches noch zwei Felder zum Anklicken besitzt. Diese Art von Meldungen mit Auswahlmöglichkeit nennt man Requester.

Wir wollen uns nun eine recht einfache Art ansehen, wie man solche Requester programmieren kann. Dazu benötigen wir zu-

erst einmal ein Fenster, in dem der Requester entstehen kann. Ein solches Fenster haben wir aber schon in unserem Beispielprogramm geöffnet.

Um nun einen Requester anzuzeigen, verwenden wir die Intuition-Funktion AutoRequest mit dem Offset -348, welche die Darstellung und Verwaltung des Requesters selbstständig überwacht. Diese Funktion benötigt folgende Parameter:

In A0 Der Zeiger auf die Fenster-Struktur, welche wir in 'windowhd' abgelegt haben.

In A1 Einen Zeiger auf die Text-Struktur des Textes, welcher über den Auswahlknöpfen stehen soll.

In A2 Wie oben für den Text des linken Knopfes.

In A3 Wie oben für den rechten Knopf.

In D0 Die IDCMP-Flags, welche angeben, welches Ereignis dem Anklicken des linken Knopfes gleichkommen soll.

In D1 Wie oben für den rechten Knopf.

In D2 Die Breite des gesamten Requesters.

In D3 Die Höhe des Requesters.

Wir fügen also in unser Programm folgende Zeilen ein:

```
AutoRequest = -348

     ...
request:
     move.l  windowhd,a0       ;Zeiger auf Fensterstruktur
     lea     btext,a1
     lea     ltext,a2          ;Zeiger auf Text-Strukturen
     lea     rtext,a3
     move.l  #0,d0             ;links nur durch Anklicken
     move.l  #0,d1             ;rechts nur durch Anklicken
     move.l  #180,d2           ;Breite und
     move.l  #80,d3            ;Höhe des Requesters
     move.l  intbase,a6        ;Intuition-Basis
     jsr     autorequest(a6)   ;Requester darstellen
     rts
```

Die in D0 und D1 übergebenen Flags bieten interessante Möglichkeiten. So sind auch die Systemmeldungen, die zum Einlegen einer bestimmten Diskette auffordern, mit dem DISKINSERTED-Flag versehen, was das Einlegen einer Diskette zum gleichen Ereignis macht wie das Anklicken von 'Retry'.

Was hier neu ist, ist der Begriff 'Text-Struktur', welcher hier immerhin dreimal auftaucht. Es handelt sich dabei wieder einmal um eine Liste, in der einige Angaben zu dem betreffenden Text eingetragen werden.

Diese Liste beginnt mit zwei Bytes, in denen die Farben definiert werden. Das erste Byte steht für die Farbe des Textes selbst, das zweite für die Hintergrundfarbe, welche jedoch hier keine Bedeutung hat.

```
btext:
        dc.b    2       ;Text-Farbe schwarz
        dc.b    0       ;Hintergrund-Farbe
```

Danach folgt ein weiteres Byte, welches den Zeichen-Modus angibt. Trägt man hier eine 0 ein, so wird der Text ganz normal ausgegeben, eine 4 bewirkt die invertierte Darstellung des Textes.

```
        dc.b    0       ;normale Textdarstellung
```

Als nächstes kommen zwei Worte in der Liste. Aus diesem Grund muß zuerst die Adresse gerade gemacht werden, indem man entweder ein weiteres Byte anhängt oder die 'even'-Direktive verwendet. Die dann folgenden Worte geben die X- und Y-Position des Textes relativ zu der oberen linken Ecke des Kastens an, in dem er dargestellt werden soll. Dieser Kasten ist für die Requester-Überschrift das Requester-Fenster, bei den Auswahl-Texten der Auswahl-Knopf.

```
        dc.w    10      ;X-Abstand
        dc.w     5      ;Y-Abstand zur oberen Ecke
```

Nun folgt ein Zeiger auf den zu verwendenden Zeichensatz. Wir setzen hier wieder eine Null ein, um den Standard-Zeichensatz zu wählen.

```
        dc.l    0       ;Standard-Zeichensatz
```

Danach müssen wir die Adresse des Textes selbst übergeben, welcher ausgegeben werden soll. Der Text muß wieder mit einem Nullbyte abgeschlossen werden.

```
        dc.l  text      ;Zeiger auf Text
```

Als Abschluß der Liste folgt nun noch ein Langwort, welches entweder auf einen weiteren Text zeigt oder Null ist, wenn kein weiterer Text benötigt wird.

```
        dc.l    0       ;kein weiterer Text
```

Hier nun die drei Text-Strukturen, die wir für unser Beispiel brauchen:

```
btext:                  ;Text-Struktur der Überschrift
 dc.b 0,1               ;Farben
 dc.b 0                 ;Modus
even
 dc.w 10,10             ;Text-Position
 dc.l 0                 ;Standard-Font
 dc.l bodytxt           ;Zeiger auf Text
 dc.l 0                 ;kein weiterer Text

bodytxt: dc.b "Requester-Text",0
even

ltext:                  ;Text-Struktur des linken Knopfes
 dc.b 0,1               ;Farben
 dc.b 0                 ;Modus
even
 dc.w 5,3               ;Text-Position
 dc.l 0                 ;Standard-Font
 dc.l lefttext          ;Zeiger auf Text
 dc.l 0                 ;kein weiterer Text
```

```
lefttext: dc.b "links",0
even

rtext:
 dc.b 0,1            ;Farben
 dc.b 0              ;Modus
even
 dc.w 5,3            ;Text-Position
 dc.l 0              ;Standard-Font
 dc.l righttext      ;Zeiger auf Text
 dc.l 0              ;kein weiterer Text

righttext: dc.b "rechts",0
even
```

Wir bekommen nach Aufruf dieses Requesters in D0 die Information zurück, welcher der beiden Knöpfe gedrückt worden ist bzw. in welchem Knopf das entsprechende Ereignis stattgefunden hat. Ist D0 Null, so war es der rechte Knopf, bei 1 war es der linke.

7.4 Ereignis-Behandlung

Stellen Sie sich vor, Sie haben ein Fenster geöffnet, welches das Schließsymbol trägt, und wollen im Programm auf die Betätigung dieses Symbols reagieren. Sie benötigen dafür ein Signal vom Intuition, welches angibt, daß ein Ereignis eingetreten ist. Dieses Signal wird im allgemeinen Message genannt, englisch für Mitteilung.

Auf welches Ereignis hin eine solche Meldung erfolgen soll, haben Sie in den IDCMP-Flags des Fensters bereits vorgegeben: Durch das Setzen des Bits für WINDOWCLOSE wird das Ereignis "Schließsymbol betätigt" für die Meldung erlaubt.

Um nun diese Meldung zu erhalten, bietet sich eine EXEC-Funktion an, und zwar die Funktion GetMsg mit dem Offset -372, welcher als Parameter die Adresse der entsprechenden Ereignis-Quelle übergeben wird. Für unseren Fall ist diese Quelle der sogenannte User-Port, der nichts mit dem User-Port eines der alten Commodore-Rechnern zu tun hat.

Dieser User-Port stellt eine Tabelle dar, in welcher die Ereignisse und deren Randbedingungen wie Mausposition und Uhrzeit des Ereignisses vom Intuition eingetragen werden.

Wie finden wir nun diesen User-Port? Dafür betrachten wir einmal den Zeiger auf die Fenster-Struktur, welchen wir aus der OpenWindow-Funktion erhalten und in 'windowhd' abgespeichert haben.

Dieser Zeiger zeigt also, wie gesagt, auf die Fenster-Struktur dieses Fensters. Diese Struktur besteht aus etlichen Einträgen, die zum Teil Kopien der Parameter aus unserer Fenster-Definitionstabelle sind. Wir wollen nicht näher auf alle dieser Einträge eingehen, da sie für uns größtenteils nicht weiter interessant sind. Wichtig ist vor allem der Zeiger auf den gesuchten User-Port, welcher in der Fenster-Struktur enthalten ist.

Es handelt sich dabei um das Langwort, welches im 86. Byte der Struktur beginnt. Wir erhalten dieses Langwort, indem wir folgendermaßen programmieren:

```
move.l  windowhd,a0      ;Zeiger auf Struktur in A0
move.l  86(a0),a0        ;User-Port-Zeiger in A0
```

Mit diesem Zeiger in A0 können wir nun die GetMsg-Funktion aufrufen, indem wir folgende Zeilen in unser Programm einfügen:

```
GetMsg  = -372

...
move.l  windowhd,a0      ;Zeiger auf Struktur in A0
move.l  86(a0),a0        ;User-Port-Zeiger in A0
move.l  ExecBase,a6      ;EXEC-Basisadresse in A6
jsr     GetMsg(a6)       ;Message holen
```

Diese Funktion kehrt sofort wieder und liefert einen Rückgabewert, wie immer im Register D0. Dieser Wert ist schon wieder ein Zeiger, welcher auf eine weitere Struktur zeigt, die Intuition-Message-Struktur. Wird in D0 jedoch eine 0 zurückgegeben, so hat kein Ereignis stattgefunden.

Das Langwort, welches ab dem 20. Byte in dieser Struktur liegt, enthält nun die Information, welches Ereignis eingetreten ist. Die Auswertung der Information ist recht einfach, da die Bits dieses Langwortes die gleiche Bedeutung haben wie diejenigen des IDCMP-Flags, welches beim Fensteröffnen bereits beschrieben wurde.

Fügen Sie nun hinter 'loop' obige Zeilen ein und setzen noch folgende Zeilen dahinter:

```
move.l  d0,a0        ;Message-Zeiger in A0
move.l  20(a0),d6    ;Ereignis in D6 retten
tst.l   d0           ;Ereignis stattgefunden?
bne     ende         ;ja!
```

Wenn Sie dieses Programm dann starten, können Sie es durch Anklicken des Schließ-Symbols wieder beenden. Auf diese Weise können wir also feststellen, ob ein Ereignis stattgefunden hat. In D6 können Sie nun außerdem feststellen, welches Ereignis es war. In unserem Beispiel steht dort die Zahl $00000200, was bedeutet, daß das Schließ-Symbol angeklickt wurde.

Um einmal auszuprobieren, ob dies auch mit anderen Ereignissen klappt, setzen Sie doch bitte einmal in der Fenster-Definitionstabelle als IDCMP-Flags anstelle der $200 eine $10200 ein. Wenn Sie diese Version dann assembliert und gestartet haben, nehmen Sie die Diskette aus dem Laufwerk: das Programm bricht ab.

Das zusätzlich gesetzte IDCMP-Flag bewirkt nämlich, daß außer dem Anklicken des Schließ-Symbols noch das Herausnehmen der Diskette gemeldet wird (DISKREMOVED). Welches der beiden möglichen Ereignisse nun stattgefunden hat, können Sie wieder an D6 erkennen. Hier steht beim Schließen des Fensters eine $200 und beim Herausnehmen der Diskette eine $10000.

7.5 Menü-Programmierung

Und nun werden wir uns einer der interessantesten Fähigkeiten des Intuition zuwenden: der Menü-Programmierung. Mit Menüs kann man ja bekanntlich sein Programm sehr komfortabel machen.

Die Möglichkeiten, Menüs darzustellen, sind ungemein vielfältig. So kann man Menüpunkte unwählbar machen, Untermenüs ausgeben und die Art der Menüeinträge wählen, ob man Texte oder Bildchen ausgeben will. Diese Vielzahl an Möglichkeiten erfordern natürlich auch etliche Parameter.

Beginnen wir am Anfang. Dargestellt wird ein Menü mittels der SetMenuStrip-Funktion des Intuition mit dem Offset -264. Diese Funktion braucht nur zwei Parameter, und zwar einen Zeiger auf die Menüstruktur des darzustellenden Menüs und den Zeiger auf die Fensterstruktur des Fensters, von dem aus das Menü funktionieren soll. Es kann ja jedes Fenster ein eigenes Menü besitzen, was aktiv wird, wenn das Fenster aktiviert wird.

Hier die Unterroutine zum Setzen des Menüs:

```
SetMenuStrip  = -264

     ...
setmenu:                        ;* Initialisieren eines Menüs
     move.l  intbase,a6         ;Intuition-Basis in A6
     move.l  windowhd,a0        ;Zeiger auf Fensterstruktur
     lea     menu,a1            ;Zeiger auf Menü-Struktur
     jsr     SetMenuStrip(a6)   ;Funktionsaufruf
     rts
```

Und natürlich auch noch die Routine zum Löschen des Menüs:

```
ClearMenuStrip  = -54

     ...
clearmenu:
     move.l  intbase,a6        ;Intuition-Basis in A6
     move.l  windowhd,a0       ;Zeiger auf Fensterstruktur
     jsr     ClearMenuStrip(a6)
     rts
```

Den Zeiger auf die Fensterstruktur haben wir bereits. Betrachten wir nun die Menüstruktur, die wir für das Menü aufbauen müssen. Eine solche Struktur muß für jedes einzelne Menü aufgebaut werden, d.h. für jeden Menütitel, der beim Betätigen der rechten Maustaste erscheint.

Diese Struktur ist wieder eine Tabelle, welche folgenden Aufbau haben muß:

Zuerst kommt ein Langwort, welches auf die Menüstruktur des nächsten Menüs zeigt. Ist das momentane Menü das letzte, so wird hier eine Null eingetragen.

```
even
menu:
     dc.l    menu1        ;Zeiger auf nächstes Menü
```

Danach folgen zwei Worte, die die X- und Y-Position des Menütitels angeben:

```
     dc.w   20            ;X-Position
     dc.w    0            ;Y-Position
```

Und nun Breite und Höhe des Menütitels in zwei Worten, angegeben in Bildschirmpunkten:

```
     dc.w   50            ;Breite
     dc.w   10            ;Höhe des Menütitels
```

Das nächste Wort beinhaltet ein Flag-Bit, welches bestimmt, ob das Menü wählbar ist oder nicht. Ein nicht wählbares Menü er-

kennt man daran, daß alle Einträge in grauer Farbe bzw. nur schwach dargestellt sind. Ist das Flag-Bit, Bit 0, gesetzt, so ist das Menü wählbar, andernfalls nicht.

```
dc.w    1           ;Menü wählbar
```

Nun folgt ein Langwort, und zwar der Zeiger auf den Text, welcher als Menütitel dargestellt werden soll. Achten Sie dabei darauf, daß die Textlänge nicht größer ist als durch die Breitenangabe erlaubt! Andernfalls entstehen unschöne Effekte.

```
dc.l    menu0text   ;Zeiger auf Titel-Text
```

Und jetzt kommt noch ein Langwort, was einen Zeiger auf die Struktur des ersten Menüeintrages dieses Menüs darstellt. Jeder Menüeintrag benötigt nämlich leider auch jeweils eine Struktur.

```
dc.l    menuitem01  ;Zeiger auf ersten Menüpunkt
```

Den Abschluß dieser Tabelle bilden noch vier Worte, die für interne Funktionen reserviert sind, jedoch dabei sein müssen.

```
dc.w    0,0,0,0     ;reservierte Worte
```

Das war die Struktur des ersten Menüs. Nach obigem Beispiel zeigt das erste Langwort auf die nächste Struktur, welche den gleichen Aufbau hat. Beim letzten Menü wird der Zeiger gleich Null gesetzt.

Was wir jetzt noch brauchen, sind die Strukturen der Menüeinträge. Diese Strukturtabellen haben diesen Aufbau:

Wieder beginnt es mit einem Zeiger auf den nächsten Menüpunkt, der beim letzten Eintrag auf Null gesetzt wird:

```
even
menuitem01:
    dc.l    menuitem02  ;Zeiger auf nächsten Menüpunkt
```

Dem folgen nun wieder vier Worte, in denen X- und Y-Position sowie die Breite und Höhe der Box eingetragen werden, in der

der Menüeintrag steht. Diese Ausmaße werden nicht direkt sichtbar, erst wenn man den Punkt mit festgehaltener rechten Maustaste mit dem Mauszeiger anwählt, sieht man die Umrisse der Box. Wie man diese sieht, wird in den Flags des nächsten Wortes bestimmt. Doch zuerst die Position und Ausmaße des Menüpunktes:

```
dc.w    0       ;X-Position des Eintrages
dc.w    0       ;Y-Position
dc.w   90       ;Breite in Bildpunkten
dc.w   10       ;Höhe in Bildpunkten
```

Die Positionsangaben sind relativ zur linken oberen Ecke des heruntergerollten Menüs anzugeben.

Das nun folgende Wort wurde bereits erwähnt: es beinhaltet Flags für die Angaben über diesen Menüpunkt. Dabei sind einige interessante Variationen möglich. Folgende Flag-Bits sind in diesem Wort enthalten:

Bit	Wert	Name	Bedeutung, wenn gesetzt
0	$0001	CHECKIT	Punkt wird bei Wahl abgehakt
1	$0002	ITEMTEXT	Text-Menüpunkt
2	$0004	COMMSEQ	Wahl auch durch Taste möglich
3	$0008	MENUTOGGLE	Haken wird ein- und ausgeschaltet
4	$0010	ITEMENABLED	Menüpunkt wählbar
6	$0040	HIGHCOMP	Punkt wird bei Wahl invertiert
7	$0080	HIGHBOX	Punkt wird bei Wahl umrahmt
8	$0100	CHECKED	Punkt ist abgehakt

Die Bedeutungen dieser Punkte im einzelnen:

CHECKIT Ist dieses Bit gesetzt, so wird automatisch ein Haken oder ein selbstdefiniertes Zeichen vor den Text gesetzt, wenn der Punkt gewählt wurde. Es sollte dabei der Text mit zwei Leerzeichen beginnen.

ITEMTEXT Der Menüpunkt ist ein normaler Text, wenn dieses Bit gesetzt ist. Andernfalls wird hier eine Zeichnung ausgegeben.

COMMSEQ Durch das Setzen dieses Bits und der Angabe eines Zeichens (s. u.) kann dieser Menüpunkt auch durch die Betätigung der rechten Amiga-Taste und der angegebenen Taste angewählt werden. Das angegebene Zeichen wird dann auch im Menü mit dem Amiga- Symbol angezeigt, wofür auch entsprechend Platz vorgesehen werden sollte.

MENUTOGGLE Ist dieses Bit gesetzt und das Abhaken erlaubt (Bit 0), so wird der Haken beim zweiten Mal anwählen dieses Punktes automatisch wieder gelöscht, beim nächsten Mal wieder gesetzt und so weiter.

ITEMENABLED Durch Löschen dieses Bits wird der Menüpunkt nicht wählbar.

HIGHCOMP Ist dieses Bit gesetzt, so wird die definierte Box um diesen Menüpunkt beim Berühren mit dem Mauszeiger invertiert.

HIGHBOX Bei diesem Modus wird die Box umrahmt, wenn sie berührt wird.

Die beiden letzten Bits geben also den Modus an, wie ein berührter Menüpunkt erscheinen soll. Folgende Kombinationen sind auch möglich:

HIGHIMAGE Sind beide Bits gelöscht, so wird bei Berührung dieser Box eine selbstdefinierte Zeichnung ausgegeben.

HIGHNONE Wenn beide Bits gesetzt sind, findet keine Reaktion auf die Berührung der Box statt.

CHECKED Dieses Bit wird sowohl vom Programm als auch von Intuition gesetzt. Es gibt an, ob vor dem Menütext ein Haken stehen soll oder nicht. Sie können also aus Ihrem Programm feststellen, ob dieser Punkt abgehakt wurde, indem Sie in diesem Wort das Bit 8 testen. Ist es gesetzt, so ist er abgehakt. Andererseits können Sie es vom Programm aus setzen, und der Punkt erscheint abgehakt.

Wir wählen die Modi CHECKIT, ITEMTEXT, COMMSEQ, MENUTOGGLE, ITEMENABLED und HIGHBOX für unser Beispiel:

```
dc.w   %10011111    ;Modus-Flags
```

Doch nun zurück zu der Struktur der Menüpunkte. Nach dem Flag-Wort folgt ein Langwort, in welchem durch Flag-Bits gewählt werden kann, ob dieser Menüpunkt einen anderen ausschließt. Wir setzen dies aber auf Null:

```
dc.l   0            ;kein Ausschluß
```

Und nun kommt der Zeiger auf die Struktur des Textes, welcher dargestellt werden soll. Ist allerdings das ITEMTEXT-Bit nicht gesetzt, so muß dieser Zeiger auf die Struktur der Zeichnung zeigen. Soll überhaupt nichts dargestellt werden, kann man diesen Zeiger auch auf Null setzen. Wir verwenden in unserem Beispiel einen Text und schreiben daher:

```
dc.l   menu01text   ;Zeiger auf Menütext-Struktur
```

Das folgende Langwort hat nur eine Bedeutung, wenn das HIGHIMAGE-Flag gesetzt ist. Dann zeigt dieses Langwort auf den Text oder die Zeichnung, welche beim Berühren der Menüpunkt-Box dargestellt werden soll. Andernfalls wird das Langwort ignoriert, weshalb wir es auf Null setzen:

```
dc.l   0            ;keine Zeichnung beim Berühren
```

Der nächste Eintrag ist ein Byte, welches für die Angabe des Tastaturzeichens verwendet wird, welches mit der rechten Amiga-Taste zusammen gedrückt diesen Menüpunkt auswählen kann. Dies gilt natürlich nur dann, wenn das COMMSEQ-Bit gesetzt ist. Wir tragen hier einmal ein Zeichen ein:

```
        dc.b    'A'         ;Anwahl auch durch Amiga/'A'
```

Da nun wieder ein Langwort folgt, muß zuerst mit 'even' die Adresse begradigt werden. Danach kommt das Langwort, welches auf die Menüpunkte-Struktur eines eventuellen Untermenüs zeigt. Diese Untermenüs werden beim Berühren des Menüpunktes mit dem Mauszeiger automatisch dort angezeigt. Eine weitere Verschachtelung ist leider nicht möglich, so daß dieses Langwort bei einem Untermenü ignoriert wird.

Will man kein Untermenü zu diesem Punkt einrichten, so muß hier eine Null eingetragen werden:

```
    even
        dc.l    0           ;kein Untermenü
```

Das nächste und letzte Langwort wird von Intuition beschrieben, wenn man mehrere Menüpunkte auswählt. In diesem Fall steht hier die Menünummer des nächsten gewählten Menüpunktes.

```
        dc.l    0           ;vorbereitet
```

Das war die Struktur für einen Menüpunkt. Was jetzt noch für diesen Punkt fehlt, ist die Textstruktur für den darzustellenden Text. Diese ist nicht sehr kompliziert, läßt aber auch einige Feinheiten in der Gestaltung des Menüs zu. Diese Textstruktur haben wir bereits bei den Requestern kennengelernt, weshalb die Erklärungen hier entfallen können.

Hier nun auf einmal die komplette Struktur eines Beispielmenüs. Es werden zwei Menüs dargestellt mit jeweils zwei Unterpunkten. Der zweite Menüpunkt des linken Menüs hat zusätzlich ein Untermenü mit zwei Einträgen. Es empfiehlt sich, diesen Programmteil wirklich abzutippen, da man damit recht schön expe-

rimentieren kann. Außerdem werden wir an Hand dieses Beispiels die Auswertung des angeklickten Menüpunktes kennenlernen.

```
;** komplette Menüstruktur für Beispiel-Menü **

menu:
     dc.l menu1          ;kein nächstes Menü
     dc.w 10,30          ;X/Y
     dc.w 50,10          ;Breite/Höhe
     dc.w 1              ;MenuEnabled
     dc.l menuname       ;Menütitel
     dc.l menuitem01     ;Menü-Einträge
     dc.w 0,0,0,0
menuname:
     dc.b "Menü 1",0     ;erster Menüname
even

menu1:
     dc.l 0              ;kein weiteres Menü
     dc.w 80,0           ;s.o.
     dc.w 50,10
     dc.w 1
     dc.l menuname1
     dc.l menuitem11
     dc.w 0,0,0,0
menuname1:
     dc.b "Menü 2",0     ;zweiter Menüname
even

menuitem01:              ;erster Menüpunkt
     dc.l menuitem02     ;Zeiger auf nächsten Eintrag
     dc.w 0,0            ;X,Y
     dc.w 130,12         ;Breite,Höhe
     dc.w $9f            ;Flags
     dc.l 0              ;Exclude
     dc.l text01         ;Zeiger auf Textstruktur
     dc.l 0              ;SelectFill
     dc.b "1"            ;Command
even
     dc.l 0              ;SubItem: keins
     dc.w 0              ;NextSelect: nein
```

```
text01:
     dc.b 0,1            ;Farben
     dc.b 0              ;Modus: überschreiben
even
     dc.w 5,3            ;X/Y-Position
     dc.l 0              ;Standard-Zeichensatz
     dc.l text01txt      ;Zeiger auf Text
     dc.l 0              ;kein weiterer Text

text01txt:
     dc.b "  Punkt 0.1",0
even

menuitem02:              ;zweiter Menüpunkt
     dc.l 0
     dc.w 0,10
     dc.w 130,12
     dc.w $57
     dc.l 0
     dc.l text02
     dc.l 0
     dc.b "2"            ;Aktivierung mit Amiga/'2'
even
     dc.l 0
     dc.w 0

text02:
     dc.b 0,1
     dc.b 0
even
     dc.w 5,3
     dc.l 0
     dc.l text02txt
     dc.l 0

text02txt:
     dc.b "  Punkt 0.2",0
even

menuitem11:              ;erster Menüpunkt des zweiten Menüs
     dc.l menuitem12     ;Zeiger auf zweiten Menüpunkt
     dc.w 0,0
     dc.w 90,12
     dc.w $52
     dc.l 0
     dc.l text11
```

```
        dc.l 0
        dc.b 0
even
        dc.l 0
        dc.w 0

text11:
        dc.b 0,1
        dc.b 0
even
        dc.w 5,3
        dc.l 0
        dc.l text11txt
        dc.l 0
text11txt:
        dc.b "Punkt 1.1",0
even

menuitem12:             ;zweiter Menüpunkt des zweiten Menüs
        dc.l 0          ;kein weiterer Punkt
        dc.w 0,10
        dc.w 90,12
        dc.w $92
        dc.l 0
        dc.l text12
        dc.l 0
        dc.b 0
even
        dc.l submenu0   ;Zeiger auf Untermenü
        dc.w 0

text12:
        dc.b 0,1
        dc.b 0
even
        dc.w 5,3
        dc.l 0
        dc.l text12txt
        dc.l 0
text12txt:
        dc.b "Punkt 1.2",0
even

submenu0:               ;Untermenü, erster Punkt
        dc.l submenu1   ;Zeiger auf nächsten Punkt
        dc.w 80,5
```

```
        dc.w 90,12
        dc.w $52
        dc.l 0
        dc.l texts0
        dc.l 0
        dc.b 0
even
        dc.l 0
        dc.w 0

texts0:
        dc.b 0,1
        dc.b 0
even
        dc.w 5,3
        dc.l 0,texts0txt,0
texts0txt:
        dc.b "U-Punkt 1",0
even

submenu1:                   ;Untermenü, zweiter Punkt
        dc.l 0
        dc.w 80,15
        dc.w 90,12
        dc.w $52
        dc.l 0
        dc.l texts1
        dc.l 0
        dc.b 0
even
        dc.l 0
        dc.w 0

texts1:
        dc.b 0,1
        dc.b 0
even
        dc.w 5,3
        dc.l 0
        dc.l texts1txt
        dc.l 0
texts1txt:
        dc.b "U-Punkt 2",0
even
```

Die einzelnen Menüpunkte dieses Beispiels haben folgende durch die Flags eingestellte Eigenschaften:

Menü 1

Der erste Punkt, 'Punkt 0.1', läßt sich auch durch die Tasten Amiga rechts und '1' auswählen. Dieser Punkt wird abwechselnd abgehakt und nicht abgehakt, wodurch sich leicht die Tastenfunktion testen läßt. Ist nämlich der Haken nicht gesetzt und man drückt diese beiden Tasten, so ist der Haken da und umgekehrt. Außerdem wird die Box dieses Punktes umrahmt, wenn der Mauszeiger sie berührt.

Der zweite Punkt, 'Punkt 0.2', läßt sich durch die Taste '2' (und Amiga rechts) anwählen. Dieser Punkt wird bei der ersten Anwahl abgehakt, dieser Haken läßt sich aber im Gegensatz zum vorherigen Punkt nicht löschen. Die Box dieses Punktes wird bei Berührung invertiert.

Menü 2

Diese beiden Punkte lassen sich nicht durch Tasten anwählen. Die Box des oberen Punktes wird bei Berührung invertiert, die untere umrahmt. Bei Berührung des zweiten Punktes, 'Punkt 1.2', wird direkt ein Untermenü daneben dargestellt, welches zwei weitere Einträge besitzt.

Sie können mit dieser Struktur nun leicht herumexperimentieren, indem Sie hier und da einige Werte oder Flags ändern. Wie Sie sehen, ist die Menüprogrammierung, abgesehen von der umfangreichen Schreiberei, nicht so kompliziert wie befürchtet und doch sehr vielseitig!

Wenn Sie nun genug experimentiert haben, wollen Sie sicher ein eigenes Programm mit Menüs erstellen. Dabei stellt sich allerdings sofort die Frage, wie das Programm feststellen kann, welcher Menüpunkt in welchem Menü betätigt wurde.

Eine erste Möglichkeit, den Menüzustand zu erfahren, wurde bereits beschrieben: den Test des CHECKED-Bits im Flags-Wort eines Menüpunktes. Wird dieses gesetzt, so hat der Benutzer mit der Maus diesen Menüpunkt angeklickt.

Dies funktioniert aber nur, wenn das Abhaken der getesteten Menüpunkte auch erlaubt war. Aber abgesehen von der Möglichkeit, alle Menüpunkte mit Häkchen zu versehen, ist es eine mühsame Programmiererei, alle Flag-Bits des gesamten Menüs der Reihe nach zu testen.

Wir haben doch schon eine Methode kennengelernt, Mitteilungen über irgendwelche Ereignisse von Intuition zu erhalten. Die Art des Ereignisses, welches da gemeldet wird, haben wir in das Register D6 gebracht und konnten dort feststellen, was passiert ist.

Wenn wir nun in dem IDCMP-Flags-Langwort der Fenster-Definition das Bit 8, MENUPICK, setzen, so wird uns von der GetMsg-Funktion auch die Anwahl eines Menüpunktes gemeldet. Wir schreiben also in unsere Schleife des Gesamtprogramms folgende Zeilen (sofern sie nicht noch dort stehen):

```
loop:
        move.l  execbase,a6       ;EXEC-Basisadresse in A6
        move.l  windowhd,a0       ;Fensterstruktur-Zeiger
        move.l  86(a0),a0         ;User-Port-Zeiger in A0
        jsr     GetMsg(a6)        ;Message holen
        tst.l   d0                ;war was ?
        beq     loop              ;kein Ereignis

        move.l  d0,a0             ;Message-Zeiger in A0
        move.l  $14(a0),d6        ;Ereignis in D6
```

Wenn das Programm an dieser Stelle angelangt ist, so hat ein Ereignis stattgefunden, dessen Flags wir in D6 vorliegen haben. Wir können nun dieses Ereignis auswerten, indem wir mit CMP oder BTST die Flag-Bits testen und, wenn sie gesetzt sind, die entsprechenden Funktionen auslösen. So können wir durch die anschließenden Zeilen:

```
cmp     #$200,d6        ;WINDOWCLOSE?
beq     ende            ;ja: Programmende
```

das Programm durch Schließen des Fensters beenden.

Hat der Benutzer nun einen Menüpunkt ausgewählt, so wird uns dies mit dem Wert $100 in D6 gemeldet. Wir müssen jetzt feststellen, welcher Menüpunkt das war.

Diese Information finden wir in einem Wort, welches direkt hinter dem Langwort mit den Ereignis-Flags in der Message-Struktur steht. Wir schreiben also:

```
move    $18(a0),d7
```

und erhalten den Code für den angeklickten Menüpunkt in D7. Wenn der Benutzer nur die rechte Taste gedrückt und losgelassen hat, ohne einen Menüpunkt ausgewählt zu haben, finden wir hier den Wert $FFFF. Dieses Wort beinhaltet sonst aber eigentlich nicht nur eine, sondern drei Informationen:

- In welchem Menü wurde gewählt?
- Welcher Menüpunkt?
- Wenn, dann in welchem Untermenü?

Diese Informationen sind in drei Bit-Gruppen verteilt, und zwar auf folgende Bits:

Bits 0 - 4 Menütitel-Nummer
Bits 5 - 10 Menüpunkt-Nummer
Bits 11 - 15 Untermenüpunkt-Nummer

All diese Nummern beginnen bei 0, d.h. der erste Menüpunkt des ersten Menüs hat die Nummern 0 und 0.

Um dies auszuprobieren, können Sie folgende Zeilen hinzufügen:

```
        move    d7,d6       ;Code in D6 bringen
        lsr     #8,d7       ;11 mal nach rechts schieben
        lsr     #3,d7       ;Untermenüpunkt ist nun in D7
        clr.l   d5
        roxr    #1,d6       ;Bit 0 in X-Flag
        roxl    #1,d5       ;Menünummer nun in D5
        and.l   #$7f,d6     ;untere Bits ausblenden
        cmp     #$7f,d6     ;kein Menüpunkt ?
        beq     loop        ;nein: weiter
        lsr     #4,d6       ;sonst Menüpunkt in D6
ende:
```

Sie können damit bei einem Testlauf mit dem SEKA leicht feststellen, ob die Auswertung auch richtig funktioniert, da Sie nach Abschluß des Programms direkt an den angezeigten Registern das Ergebnis ansehen können.

Wenn Sie nun ein Programm mit z.B. 4 Menüs mit jeweils 10 Menüpunkten schreiben wollen, ist für diese 44 Tabellen die obige Schreibweise viel zu aufwendig. Aus diesem Grund möchte ich Ihnen ein kleines Programm anbieten, welches die nötigen Strukturtabellen selbständig herstellt.

Die so entstandene Menüstruktur ist allerdings nur sehr einfach aufgebaut, d.h. ohne spezielle Auswahlmodi oder Untermenüs. Wenn Sie dennoch solche Extras haben wollen, so können Sie das Programm dennoch verwenden, müssen aber nachher mit einigen MOVE-Befehlen die gewünschten Flags bzw. Zeiger nachträglich eintragen.

Die Angaben, die auch dieses Programm natürlich benötigt, sind lediglich die Namen der Menüs und der Menüpunkte, die im Menü stehen sollen. Die Adressen dieser Menütexte werden dann in einer Tabelle abgelegt, welche folgenden sehr einfachen Aufbau hat:

```
dc.l    Menütitel 1
dc.l    Punkt1, Punkt2, Punkt3, ...,0
dc.l    Menütitel 2
dc.l    Punkt1, Punkt2, Punkt3, ...,0
dc.l    Menütitel 3  oder 0
```

Das Programm ist so ausgelegt, daß es bis zu 4 Menüs nebeneinanderlegen kann (bei normaler Bildschirmauflösung), was oft völlig ausreicht. Die obige Tabelle endet damit, daß anstelle des Zeigers auf den nächsten Menütitel eine Null eingetragen wird. Sie sehen, dies ist schön einfach!

Das Programm wird einfach in unserem Gesamtprogramm eingefügt, und zwar direkt hinter dem Label 'setmenu'. So wird durch den Aufruf 'bsr setmenu' die Menüstruktur aufgebaut und auch gleich initialisiert. Für das restliche Programm ändert sich somit nichts, es wird nur kürzer.

Hier nun der Programmteil als komplette 'setmenu'-Routine:

```
setmenu:                        ;* Menüstruktur initialisieren
     lea     mentab,a0          ;Zeiger auf Textzeiger in A0
     lea     menu,a1            ;Zeiger auf Menüfeld in A1
     move    #10,d1             ;horizontale Menüposition=10

menuloop:
     clr.l   d2                 ;vertikale Menüpunkt-Position =0
     move.l  a1,a2              ;Adresse für Zeiger retten
     tst.l   (a0)               ;noch ein Menü da ?
     beq     setmenu1           ;nein: Ende
     clr.l   (a1)+              ;'kein weiteres Menü' vorbereiten
     move    d1,(a1)+           ;X-Position setzen
     add.l   #70,d1             ;und erhöhen
     move.l  #50,(a1)+          ;Y-Position und Breite
     move.l  #$a0001,(a1)+      ;Höhe und Flag
     move.l  (a0)+,(a1)+        ;Menütitel
     lea     12(a1),a3
     move.l  a3,(a1)+           ;Zeiger auf Menüpunkte
     clr.l   (a1)+              ;reservierte Worte
     clr.l   (a1)+

itemloop:
     tst.l   (a0)               ;letzter Eintrag ?
     beq     menuend            ;ja: Menü fertig
     lea     54(a1),a3
     move.l  a3,(a1)+           ;Zeiger auf nächsten Punkt
     move.l  d2,(a1)+           ;X/Y-Position
     add     #10,d2             ;Y-Position +10
     move.l  #$5a000a,(a1)+     ;Breite/Höhe
     move    #$52,(a1)+         ;Flag: normal
```

```
        clr.l   (a1)+           ;kein Ausschluß
        lea     16(a1),a3
        move.l  a3,(a1)+        ;Textstruktur-Zeiger
        clr.l   (a1)+           ;keine Füllstruktur
        clr.l   (a1)+           ;kein Command, kein Untermenü
        clr.l   (a1)+           ;und keine Fortsetzung

        move    #$1,(a1)+       ;Textstruktur setzen: Farben
        clr     (a1)+           ;Modus 0
        move.l  #$50003,(a1)+   ;X/Y-Position
        clr.l   (a1)+           ;Standard-Zeichensatz
        move.l  (a0)+,(a1)+     ;Textzeiger
        clr.l   (a1)+           ;keine Fortsetzung

        bra     itemloop        ;nächster Punkt...

menuend:                        ;evtl. Übergang zum nächsten Menü
        clr.l   -54(a1)         ;Zeiger auf nächsten Punkt löschen
        tst.l   (a0)+           ;Tabellenzeiger erhöhen
        tst.l   (a0)            ;noch ein Menü da ?
        beq     setmenu1        ;nein: fertig
        move.l  a1,(a2)         ;Zeiger auf nächstes Menü
        bra     menuloop        ;und weitermachen

setmenu1:                       ;* Menü initialisieren (wie vorher)
        move.l  intbase,a6      ;Intuition-Basisadresse in A6
        move.l  windowhd,a0     ;Fensterstruktur in A0
        lea     menu,a1         ;Zeiger auf Menüstruktur
        jsr     SetMenuStrip(a6)
        rts
```

Zu diesem Programm gehören noch drei Dinge: der zu verwendende Speicherplatz für die Struktur, die Tabelle der Textzeiger und die Texte selbst. Hier ein Beispiel:

```
mentab:
        dc.l menu1              ;erster Menütitel
        dc.l mp11,mp12,mp13     ;Menüpunkte
        dc.l 0                  ;Menü 1 Ende

        dc.l menu2              ;zweiter Menütitel
        dc.l mp21,mp22,mp23     ;Menüpunkte
        dc.l 0                  ;Menü 2 Ende

        dc.l 0                  ;Ende des gesamten Menüs!
```

```
;**  Menü-Texte  **

menu1: dc.b "MenÜ 1",0
mp11:  dc.b "Punkt11",0
mp12:  dc.b "Punkt12",0
mp13:  dc.b "Punkt13",0

menu2: dc.b "MenÜ 2",0
mp21:  dc.b "Punkt 21",0
mp22:  dc.b "Punkt 22",0
mp23:  dc.b "Punkt 23",0
even

;**  Platz für Menüstruktur  **

menu:  blk.w 500
```

Achten Sie hierbei bitte darauf, daß der reservierte Speicherplatz für die Menüstruktur groß genug ist und ändern Sie gegebenenfalls die Angabe 'blk.w 500' in den berechneten Wert.

Wenn Sie dieses Programm verwenden und dennoch einige Besonderheiten in das Menü einbauen wollen (z.B. Tastatur-Kommandos), so können Sie nach Ablauf des Programms in die entstandene Menüstruktur-Tabelle zusätzliche Eintragungen machen. Sie finden das entsprechende Wort (bzw. Byte oder Langwort) in der Tabelle, wenn Sie folgendermaßen vorgehen:

Um z.B. das Tastatur-Kommando-Byte des 2. Eintrages im 1. Menü zu finden, müssen Sie rechnen:

```
Adresse = Anfangsadresse + Menü*30 + (Eintrag-1)*54 + 26
```

das heißt in unserem Beispiel:

```
Adresse = menu + 30 + 54 + 26
        = menu + 110
```

Die 26 ist der Abstand vom Anfang der MenuItem-Struktur zu dem gesuchten Byte, dem Kommando-Byte. Auf diese Weise können Sie durch eine Reihe von MOVE-Befehlen das fertige

Menü an Ihre Wünsche anpassen. Übrigens: bei obigem Beispiel muß natürlich noch das entsprechende Flag-Bit gesetzt werden, damit das Tastatur-Kommando auch anerkannt wird!

Wenden wir uns nun wieder dem Fenster zu. Es ist zwar ganz nett, ein Fenster zu haben, welches man verändern oder auch schließen kann, aber man sollte ja auch in diesem Fenster Text ausgeben können!

7.6 Textausgabe

Die Textausgabe-Funktion des Intuition ist mit unseren schon erworbenen Kenntnissen sehr einfach. Wir brauchen nur die Funktion PrintIText mit dem Offset -216 aufzurufen und ihr einige Parameter zu übergeben. Die Parameter sind:

In A0 Einen Zeiger auf den RastPort des Fensters, welchen wir in der Fensterstruktur finden können.

In A1 Einen Zeiger auf die Textstruktur des Textes, welcher ausgegeben werden soll.

In D0 Die X-Position.

In D1 Die Y-Position des Textes im Fenster.

Die Angaben über die X- und Y-Position sind einfach zu machen. Auch die Textstruktur, die wir für den Text aufbauen müssen, ist kein Geheimnis mehr, da wir sie bereits zweimal verwendet haben (Requester und Menüs).

Neu ist hier nur der Begriff des RastPorts des Fensters. Dieser RastPort stellt eine Art Zusatzstruktur dar, welche das entsprechende Fenster beschreibt. Diese Adresse wird von einigen Funktionen des Intuition benötigt.

Der Zeiger auf den RastPort liegt in der Fensterstruktur ab dem 50ten Byte. Wir schreiben also

```
        move.l  windowhd,a0     ;Adresse der Fensterstruktur
        move.l  50(a0),a0       ;RastPort-Adresse in A0
```

und haben die Adresse des RastPorts ermittelt. Für die Ausgabe eines Textes brauchen wir also nur eine Routine, die etwa so aussieht, wenn wir die X- und Y-Position in D0 und D1 sowie die Adresse der Textstruktur in A1 übergeben:

```
PrintIText   = -216

     ...
print:
        move.l  intbase,a6      ;Intuition-Basisadresse in A6
        move.l  windowhd,a0     ;Adresse der Fensterstruktur
        move.l  50(a0),a0       ;RastPort-Adresse in A0
        jsr     PrintIText(a6)  ;Funktion aufrufen
        rts
```

Sie können diese Routine auch gleich einmal ausprobieren, indem Sie z.B. den Text des Requesters, für den ja eine Struktur noch im Programm vorhanden ist, ausgeben lassen. Dafür schreiben Sie dann vor dem Label 'loop' folgende Zeilen:

```
        lea     btext,a1        ;Zeiger auf Textstruktur in A1
        move.l  #10,d0          ;X-Position
        move.l  #30,d1          ;Y-Position des Textes
        bsr     print           ;Text ausgeben
```

Und schon steht der Text mitten auf dem Fenster. Sollte er nicht erscheinen, so überprüfen Sie erst einmal die in der Textstruktur angegebene Farbe des Textes, ob diese nicht 0 ist. Ändern Sie dann dieses Byte in eine 3, und der Text wird beim nächsten Programmstart rot erscheinen.

7.7 Images

Image ist die Bezeichnung für eine Zeichnung, welche in einem rechteckigen Feld liegt und bitweise definiert wird. Beispiele für solche Images sind z.B. die Diskettensymbole auf dem Intuition-Screen oder auch die Systemgadgets in den Screen- und Fenster-Rändern, welche beim Anklicken Funktionen auslösen können.

Das Rechteck, innnerhalb dessen die Zeichnung liegt, kann beliebig groß sein. Da aber für jeden Bildpunkt innerhalb dieses Rechtecks je ein Bit benötigt wird, ist die Programmierung von Bildschirmgroßen Images nicht gerade ratsam. Beschränken wir uns daher auf etwa 32x16 Bit, was Images von ungefähr 3x1 cm zuläßt.

Solche Images lassen sich auf vielfältige Art und Weise darstellen, wie Sie schon bei den Gadgets in den Fenstern sehen können. Es gibt natürlich auch eine Intition-Funktion, welche das Zeichnen eines Images übernimmt.

Es handelt sich dabei um die DrawImage-Funktion mit dem Offset -114, welche 4 Parameter benötigt:

In A0 die Adresse des RastPorts, in dem das Image gezeichnet werden soll. Wie man diese Adresse bekommt, haben wir schon bei der Text-Funktion kennengelernt.

In A1 kommt die Adresse der Struktur des darzustellenden Images.

In D0 kommt die relative X-Position.

In D1 kommt die relative Y-Position der Zeichnung.

Gehen wir davon aus, daß wir nur ein Fenster haben, können wir wieder eine einfache Unterroutine schreiben, der wir in A1 die Adresse der Image-Struktur und in D0 und D1 die Position übergeben, an die die Zeichnung gebracht werden soll:

```
DrawImage  = -114

     ...
draw:                          ;* Image zeichnen
     move.l  intbase,a6        ;Intuition-Basisadresse in A6
     move.l  windowhd,a0       ;Zeiger auf Fensterstruktur
     move.l  50(a0),a0         ;nun RastPort-Adresse in A0
     jsr     DrawImage(a6)     ;Image zeichnen
     rts
```

Wir brauchen jetzt die erwähnte Struktur des Images. Diese Struktur besteht aus 9 Einträgen folgender Bedeutung:

Die ersten beiden Einträge sind zwei Worte, die den Abstand in X- und Y-Richtung von den angegebenen Koordinaten angeben, in dem das Image gezeichnet werden soll. Wir setzen hier einfach zwei Nullen ein:

```
image:
     dc.w    0,0          ;X- und Y-Abstand
```

Danach folgen zwei Worte, die die Breite und die Höhe des Images in Bildschirmpunkten angeben. Angenommen, wir wollen ein Image aus 32x13 Punkten zeichnen, so geben wir an:

```
     dc.w    32,13        ;Breite und Höhe des Images
```

Das nächste Wort der Liste gibt an, wieviele Planes die Zeichnung haben soll. Wird ein einfaches Image, welches nur in zwei Farben dargestellt werden soll, gewünscht, so gibt man hier eine 1 an, bei mehr Farben entsprechend mehr. Wenn mit mehreren Farben gearbeitet werden soll, müssen natürlich auch für das Bitmuster des Images entsprechend mehr Daten im eigentlichen Bitmuster angegeben werden. Bleiben wir daher vorerst bei einer 1:

```
     dc.w    1            ;eine Bitplane: 2^1=2 Farben
```

Nun folgt ein Langwort, welches auf die eigentlichen Daten des Images zeigen:

```
dc.l    imgdata     ;Zeiger auf Image-Daten
```

Weiter geht es mit zwei Bytes, deren Bedeutungen sehr interessant sind. Das erste Byte namens PlanePick gibt an, in welchen Planes des Fensters oder Screens, in dem gezeichnet wird, die Daten des Images erscheinen sollen. Da wir von nur einer darzustellenden Plane ausgehen, ist hier also die Angabe einer Bit-Plane des Fensters gefragt. Diese Information liegt in den Bits dieses Bytes, wobei Bit 0 für Plane 0, Bit 1 für Plane 1 usw. steht. Mit dieser Angabe wird somit auch die Farbe definiert, in der das Image dargestellt werden soll. Geben wir also eine 2 an, so wird für jedes gesetzte Bit unseres Images ein roter Punkt gesetzt.

```
dc.b    1           ;Zeichnung rot: Plane 1
```

Das zweite Byte mit dem Namen PlaneOnOff ist hierzu eine interessante Ergänzung. Hier entspricht ebenfalls jedes Bit der Fenster-Bit-Plane gleicher Nummer. Interessant sind dabei aber nur diejenigen Bits, bei denen im PlanePick-Byte dieses Bit gelöscht ist. Ist das Bit in PlaneOnOff dann gesetzt, so wird jedes Bit des Images in der entsprechenden Plane gesetzt, andernfalls gelöscht. Um also zu erreichen, daß jedes nicht gesetzte Bit des Images in weiß dargestellt wird, geben wir hier eine 1 an: alle Bits des Images, welche nicht gesetzt sind, werden in Plane 1 gesetzt und damit weiß.

```
dc.b    1           ;Hintergrund: weiß
```

Als letzter Eintrag der Struktur kommt nun ein Langwort, welches auf ein eventuell zusätzliches Image zeigt. Wir brauchen dies nicht und setzen dieses Langwort daher auf Null:

```
dc.l    0           ;kein weiteres Image
```

Hier noch einmal die ganze Image-Struktur auf einen Blick:

```
image:
    dc.w    0,0         ;X- und Y-Abstand
    dc.w    32,13       ;Breite und Höhe des Images
    dc.w    1           ;eine Bitplane: 2^1=2 Farben
    dc.l    imgdata     ;Zeiger auf Image-Daten
    dc.b    1           ;Zeichnung rot: Plane 1
    dc.b    1           ;Hintergrund: weiß
    dc.l    0           ;kein weiteres Image
```

Nun zu den Image-Daten selbst. Für jede Reihe von Punkten wird durch ein Wort, Langwort oder mehrere davon das Muster angegeben, indem für zu setzende Punkte des Images das entsprechende Bit gesetzt wird. Dies wird nun sooft wiederholt, wie durch die Höhe des Images vorgegeben worden ist. Die Daten jeder Zeile müssen immer an Wortgrenzen, also an geraden Adressen beginnen.

Für unser Beispiel ist es einfach, die Daten unterzubringen, da wir 32 Punkte horizontal vorgegeben haben, was genau einem Langwort entspricht. Um das Image zu programmieren, bietet sich hier die Binärdarstellung der Daten an.

Nehmen wir als Beispiel ein Image, was einen Schalter in 'OFF'-Stellung darstellt. Diese Form ist aus gutem Grund gewählt, weshalb Sie sie wirklich abtippen sollten. Wir werden nämlich in dem späteren Kapitel über Gadgets sehen, wie wir diesen Schalter umschalten können. Hier die Beispieldaten des Schalter-Images:

```
imgdata:                ;Daten für Schalter in OFF-Stellung
 dc.l %00000000000000000000000000000000
 dc.l %00000000000000000000111000000000
 dc.l %00011101110111000001111100000000
 dc.l %00010101000100000001111100000000
 dc.l %00010101100110000001111000000000
 dc.l %00011101000100000011100000000000
 dc.l %00000000000000000011100000000000
 dc.l %00000000000000000111000000000000
 dc.l %00000000000111111111100000000000
 dc.l %00000000001111111111110000000000
 dc.l %00000000001111111111110000000000
 dc.l %00000000000110000001100000000000
 dc.l %00000000000000000000000000000000
```

Wenn Sie diese Daten eingetippt haben, können Sie nun die Darstellung ausprobieren. Geben Sie dafür vor dem 'loop'-Label ein:

```
move.l  image,a1        ;Zeiger auf Image-Struktur
move    #30,d0          ;X-Position im Fenster
move    #50,d1          ;Y-Position
bsr     draw            ;Image zeichnen
```

Nun, wie gefällt Ihnen das Ergebnis, welches auf dem Bildschirm erscheint? Diesen Schalter werden wir wieder antreffen, wenn es um das Umschalten des Schalters in die 'ON'-Stellung geht, bei den Gadgets. Doch vorher müssen wir uns noch mit einer anderen Methode beschäftigen, mit der man in Fenstern zeichnen kann.

7.8 Borders

Border bedeutet auf deutsch soviel wie Rahmen, was für diese Funktion ein etwas ungenauer Ausdruck ist. Mit dieser Funktion lassen sich nämlich beliebig viele zusammenhängende Linien zeichnen, egal welcher Länge oder unter welchem Winkel.

Diese Funktion wird von Intuition eingesetzt, um z.B. die Umrahmungen von Fenstern und Screens zu zeichnen. Wir werden sie ebenfalls brauchen, um Umrandungen von verschiedenen Gebilden oder Texten herzustellen, besonders wenn wir zu den sogenannten String-Gadgets kommen. Aber dazu später.

Die Bedienung dieser Funktion ist recht einfach. Wir brauchen zunächst einmal die Intuition-Funktion DrawBorder mit dem Offset -108, welcher wir 4 Parameter übergeben:

In A0 die RastPort-Aresse des Ausgabemediums, in dem die Linien gezeichnet werden sollen. Wir verwenden hierfür wieder unser Fenster.

In A1 wird die Adresse der Border-Struktur übergeben, welche wir uns gleich ansehen werden.

In D0 kommt die relative X-Koordinate, ab der die in der X-/Y-Koordinatenliste für die Linien Positionswerte berechnet werden.

In D1 die relative Y-Koordinate. Relativ heißt auch hier wieder, daß sich diese Angaben auf die linke obere Ecke des Fensters beziehen.

Wir wollen uns hierfür wieder eine kleine Unterroutine schreiben, welcher wir beim Aufruf in A1 die Strukturadresse und in D0 und D1 die X/Y-Koordinaten übergeben müssen. Gezeichnet wird dann grundsätzlich in das Fenster, dessen Strukturadresse in 'windowhd' steht.

```
DrawBorder  = -108

     ...
borderdraw:                    ;* Mehrfachlinie zeichnen
     move.l  intbase,a6        ;Intuition-Basisadresse in A6
     move.l  windowhd,a0       ;Zeiger auf Fensterstruktur
     move.l  50(a0),a0         ;nun RastPort-Adresse in A0
     jsr     DrawBorder(a6)    ;Linien zeichnen
     rts
```

Doch nun zu der benötigten Struktur des Borders. Es werden 8 Parameter in dieser Liste aufgeführt, und zwar folgende:

Zuerst werden in zwei Worten der vertikale und der horizontale Abstand von den im Funktionsaufruf angegebenen Koordinaten angegeben. Um die Übersicht über die diversen Abstandsangaben nicht zu verlieren, tragen wir hier jeweils eine Null ein:

```
border:
     dc.w    0                 ;horizontaler Abstand
     dc.w    0                 ;vertikaler Abstand
```

Danach folgen wieder zwei Bytes, in denen die Farben bestimmt werden. Wir wählen hier für einen roten Rahmen:

```
     dc.b    3                 ;roter Rahmen
     dc.b    0                 ;Hintergrund (unbenutzt)
```

Wie Sie sehen, ist die Angabe der Hintergrundfarbe nicht von Bedeutung. Dennoch gibt es die Möglichkeit, zwischen zwei Modi zu wählen, in denen die Linien gezeichnet werden können. Diesen Modus bestimmt das folgende Byte. Ist es eine 0, so wird jede Linie vollständig in der angegeben Farbe gezeichnet, unabhängig davon, was vorher dort war. Diesen Modus nennt man JAM1. Die andere Möglichkeit, bei der beide Farbangaben ignoriert werden, ist der XOR-Modus. Hierbei werden alle Punkte, die unter der Linie liegen, im Farbwert invertiert. So wird aus einem weißen ein schwarzer oder aus einem blauen ein roter Punkt. Dieser Modus wird durch eine 2 eingestellt. Wir verwenden in unserem Beispiel aber lieber den JAM1-Modus:

```
dc.b    0               ;Modus: JAM1 (2=XOR)
```

Nun wird in einem weiteren Wort die Angabe gemacht, wie viele Koordinatenpaare in der Liste vorliegen. Da dieses Wort wieder an einer geraden Adresse liegen muß, wird zuerst eine 'even'-Anweisung eingegeben, danach die Anzahl der Paare. Beachten Sie bitte, daß für das Zeichnen von zwei Linien drei Punkte nötig sind: Anfangs-, Eck- und Endpunkt. Um also einen rechteckigen Rahmen zu zeichnen, brauchen wir 5 Paare:

```
dc.b    5               ;5 XY-Paare sind zu verbinden
```

Als nächstes folgt der Zeiger auf die Koordinatentabelle, in der die zu verbindenden Punkte definiert werden:

```
dc.l  koord             ;Zeiger auf Koordinatentabelle
```

Den Abschluß der Border-Struktur bildet ein Langwort, welches auf eine weitere Border-Struktur zeigen kann. Hat man keine weitere Struktur vorliegen, wird hier eine Null eingetragen. Der Zeiger ist durchaus nützlich, da man durch die Verbindung zweier eigenständiger Border-Strukturen z.B. einen zweifarbigen Rand um etwas herumzeichnen und es somit gut hervorheben kann. Wir verwenden diesen Zeiger jedoch in unserem Beispiel nicht:

```
dc.l    0               ;keine weitere Struktur
```

Das war die Border-Struktur. Was nun noch fehlt, ist die Koordinatenliste selbst. Hier für unser Beispiel die aus 5 Paaren bestehende Liste, welche ein Rechteck zeichnet. Ich empfehle Ihnen, diese Werte zu übernehmen, da wir sie in einem späteren Beispielprogramm noch brauchen werden.

```
koord:                          ;Koordinaten für Rechteck-Rahmen
     dc.w    -2,-2
     dc.w    80,-2
     dc.w    80, 9
     dc.w    -2, 9
     dc.w    -2,-2
```

Hier noch einmal die gesamte Border-Struktur auf einen Blick:

```
border:
     dc.w      0             ;horizontaler Abstand
     dc.w      0             ;vertikaler Abstand
     dc.b      3             ;roter Rahmen
     dc.b      0             ;Hintergrund (unbenutzt)
     dc.b      0             ;Modus: JAM1 (2=XOR)
     dc.b      5             ;5 XY-Paare sind zu verbinden
     dc.l  koord             ;Zeiger auf Koordinatentabelle
     dc.l      0             ;keine weitere Struktur

koord:                       ;Koordinaten für Rechteck-Rahmen
     dc.w    -2,-2
     dc.w    80,-2
     dc.w    80, 9
     dc.w    -2, 9
     dc.w    -2,-2
```

Wenn Sie all dies eingegeben haben, können Sie das ganze jetzt ausprobieren. Dazu schreiben Sie vor den 'loop'-Label des Programms:

```
     lea     border,a1       ;Adresse der Border-Struktur
     move    #20,d0          ;X-Grundposition
     move    #80,d1          ;Y-Grundposition
     bsr     borderdraw      ;Rahmen zeichnen
```

Sie sehen, mit dieser Funktion kann man durch eine entsprechende Anzahl von X/Y-Koordinaten auch den Eiffelturm zeichnen lassen. Doch nun genug der einfachen Zeichnungen, wir wollen nun unsere Zeichnungen und Texte zum Leben erwecken bzw. mit der Maus manipulierbar machen!

7.9 Gadgets

Der Begriff Gadgets ist bereits bei dem Aufbau von Fenstern aufgetaucht. Hierbei handelte es sich bisher um System-Gadgets wie das Schließsymbol des Fensters, welches man durch Anklicken aktivieren kann und damit evtl. eine Funktion des Programms auslöst.

Zusätzlich zu diesen festgelegten Gadgets kann man auch eigene entwerfen und darstellen. Dabei bietet das Intuition eine enorme Menge an Möglichkeiten, mit denen man die interessantesten Effekte zaubern kann!

Man unterscheidet zunächst einmal vier Gadget-Typen:

- Boolean-Gadgets dienen dazu, zwischen Ja oder Nein zu wählen, indem man sie anklickt und damit aktiviert (Ja) oder desaktiviert (Nein).
- String-Gadgets, die zur Eingabe von Texten vorgegebener Länge verwendet werden können.
- Integer-Gadgets sind eine Sonderform von String-Gadgets und dienen zur Eingabe einer Dezimalzahl, deren Wert das Intuition selbstständig in ein Langwort umrechnet und dem Programm zurückgibt.
- Proportional-Gadgets schließlich lassen die Auswahl eines analogen Wertes mit der Maus zu, indem man etwas hin- und herschiebt.

7.9.1 Boolean-Gadgets

Beginnen wir mit dem einfachsten Typ, dem Boolean-Gadget. Ein solches Gadget ist z.B. auch das Schließ-Symbol der Fenster, welches ja auch nur zwischen angeklickt und nicht angeklickt unterscheiden kann. Wir wollen nun mit diesem Typ Schritt für Schritt ein Gadget entwickeln. Die Flags usw. sind für die anderen Gadgets weitgehend gleich.

Jedes Gadget benötigt eine Struktur mit 15 Einträgen. Der Zeiger auf diese Struktur wird dann in die Struktur des Fensters, Screens oder auch des Requesters eingetragen, in dem das Gadget erscheinen soll. Für diesen Zweck war ja dort jeweils ein Langwort vorgesehen, in das wir bisher immer eine Null eingeschrieben haben. Wird dort die Adresse der Gadget-Struktur eingetragen, so wird beim Öffnen des Fensters das (oder die) Gadget(s) gleich mit dargestellt.

Die Einträge in die Gadget-Struktur sind in folgender Reihenfolge einzusetzen:

Das erste Langwort ist ein Zeiger, der ggf. auf das nächste zu installierende Gadget zeigt. Die Gadgets sind also wie Perlen auf der Schnur aufgereiht, wobei der Zeiger, der in die Window-Struktur einzutragen ist, auf das erste Gadget dieser Kette zeigen muß. Wir wollen zunächst nur ein Gadget in unser Fenster bringen, so daß wir hier eine Null eintragen:

```
gadget1:
        dc.l    0               ;kein weiteres Gadget
```

Nun folgen wieder einmal zwei Worte, welche die Position des Gadgets auf dem Fenster bestimmen. Hierfür gibt es allerdings einige Möglichkeiten der Positionsbestimmung, welche wir in den danach folgenden Flags bestimmen. Gehen wir aber erst einmal von einem fest positionierten Gadget aus:

```
        dc.w    40              ;X-Position und
        dc.w    50              ;Y-Position des Gadgets
```

Danach folgen zwei Worte, welche die Ausmaße der 'Hit-Box' des Gadgets bestimmen. Diese Box gibt nicht etwa die sichtbare Größe des Gadgets an (die hängt schließlich von den eventuell angegebenen Image-Daten ab), sondern des Rechtecks, welches von Intuition überwacht wird. Wird nämlich der Mauszeiger in diese Box gebracht und der linke Knopf betätigt, so wird das Gadget aktiviert. Das Anklicken von Teilen des Gadgets außerhalb dieser Box wird nicht beachtet!

```
dc.w   32          ;Breite und
dc.w   13          ;Höhe der Hit-Box
```

Nun folgt das Wort, welches in seinen Bits die Eigenarten des Gadgets bestimmt. In den Bits 0 und 1 wird bestimmt, was mit dem Gadget geschehen soll, wenn es (bzw. seine Hit-Box) angeklickt wird. Die Bedeutung des Wertes in diesen Bits ist folgende:

Bit 0,	1	Wert	Name	Bedeutung
0	0	0	GADGHCOMP	das Gadget wird invertiert
0	1	1	GADGHBOX	das Gadget wird umrandet
1	0	2	GADGHIMAGE	es erscheint ein anderes Image
1	1	3	GADGHNONE	keine Reaktion

Bit 2 bestimmt, ob das Gadget aus einer Zeichnung oder einem Rahmen bestehen soll: ist es gesetzt (Wert +4), handelt es sich um ein Image, andernfalls ist es ein Border.

Das nächste Bit bestimmt, ob sich die Position des Gadgets auf den oberen oder den unteren Rand des Fensters beziehen soll. Ist es gesetzt (Wert +8), wird die Position relativ zum unteren, andernfalls relativ zum oberen Rand gewertet. Das nächste Bit hat die gleiche Bedeutung für die horizontale Position: ist es gesetzt (Wert +$10), ist es eine relative, andernfalls eine absolute Positionierung.

Beachten Sie hierbei, daß bei der Definition eines relativen Gadgets die Positionsangaben in den ersten Worten der Struktur negativ sein müssen, da die gewünschte Position dann nicht unter, sondern über der Bezugsposition liegt!

Somit hat man die Möglichkeit, sowohl eine absolute als auch eine relative Position des Gadgets zu bestimmen. Ein Beispiel für ein absolut positioniertes Gadget ist das System-Gadget für das Schließen des Fensters, ein relatives Gadget ist das Symbol zur Größenveränderung.

Auch für die Breite und die Höhe des Gadgets bzw. der Hit-Box kann relativ zu Fenstergröße bestimmt werden. Dies wird mit Bit 5 für die Breite (Wert +$20) und Bit 6 für die Höhe (Wert +$40) angegeben. Auch hier gilt, daß ein gesetztes Bit die relative Größe bedeutet.

Bit 7 (Wert +$80) bewirkt, wenn es gesetzt ist, daß das Gadget sofort aktiv ist, wenn das Fenster geöffnet wird.

Bit 8 (Wert +$100) bestimmt, ob das Gadget überhaupt wählbar ist. Ist dieses Bit gesetzt, so kann das Gadget nicht aktiviert werden.

Wir wählen für unser Beispiel die absolute Positionierung und Größe, die invertierte Erscheinung des aktivierten Gadgets und die Darstellung des Gadgets als Image. Dies bedeutet, daß wir den Wert 4 eintragen müssen:

```
dc.w    $4          ;Flags: Image, invertieren
```

Nun folgt wieder ein Wort, dessen Bits jeweils Flags sind. Diese Flags nennen sich Activation Flags und bestimmen einige Funktionen des Gadgets. Die Bits und deren Wert und Bedeutung sind folgende:

Bit	Wert	Name	Bedeutung: Nachricht bei
0	1	RELVERIFY	bewirkt, daß das Gadget nur wirklich aktiviert wird, wenn der linke Mausknopf auch über dem Gadget losgelassen wurde
1	2	GADGIMMEDIATE	läßt das Gadget sofort aktivieren, wenn es angeklickt wird.
2	4	ENDGADGET	läßt, wenn das Gadget in einem Requester liegt, diesen bei Anwahl beenden und somit verschwinden.
3	8	FOLLOWMOUSE	läßt bei selektiertem Gadget ständig die Mausposition melden, bis es deselektiert wird. Hiermit läßt sich das Verschieben eines Gadgets mit der Maus realisieren, wenn man die Gadget-Position entsprechend korrigiert.
4	$10	RIGHTBORDER	bewirken bei der Verwendung von Bordern, daß
5	$20	LEFTBORDER	die entsprechende Seite der Größe des Gadgets
6	$40	TOPBORDER	jeweils angepaßt wird, damit es in den Rand
7	$80	BOTTOMBORDER	hineinpaßt.
8	$100	TOGGLESELECT	läßt das Gadget bei jedem Anklicken seinen Zustand umdrehen: aus aktiviert wird desaktiviert und umgekehrt.
9	$200	STRINGCENTER	bestimmen bei String-Gadgets, wie der String er-
10	$400	STRINGRIGHT	scheinen soll, mittig oder rechtsbündig. Ist keines dieser Bits gesetzt, wird der String linksbündig ausgegeben.
11	$800	LONGINT	macht aus einem String-Gadget ein Integer-Gadget (Erklärung später).
12	$1000	ALTKEYMAP	bewirkt, daß bei der Eingabe eines String-Gadgets eine andere Tastaturbelegung gilt.

Dies waren die Activation-Flags. Wir wählen nun die Flags TOGGLESELECT und GADGETIMMEDIATE für unser Beispiel:

```
        dc.w  $102      ;Activation
```

Das nächste Wort der Gadget-Struktur bestimmt den Gadget-Typ. Hier die Bedeutung der einzelnen Bits:

Bit	Wert	Name	Bedeutung: Nachricht bei
0	1	BOOLGADGET	dies ist ein Boolean-Gadget
1	2	GADGET002	?
2	4	STRGADGET	String- oder Integer-Gadget
0+1	3	PROPGADGET	Proportional-Gadget

System-Gadgets:

4	$10	SIZING	Größenveränderungs-Gadget
5	$20	WDRAGGING	Verschiebe-Gadget für Fen-ster
4+5	$30	SDRAGGING	Selbiges für Screens
6	$40	WUPFRONT	Gadget zum 'Fenster nach vorn holen'
6+4	$50	SUPFRONT	Gadget zum 'Screen nach vorn holen'
6+5	$60	WDOWNBACK	'Fenster nach hinten bringen'
6+5+4	$70	SDOWNBACK	'Screen nach hinten bringen'
7	$80	CLOSE	Fenster-Schließ-Gadget

Typen-Definitionen:

12	$1000	REQGADGET	Requester-Gadget
13	$2000	GZZGADGET	Rand-Gadget in GIMME-ZEROZERO-Fenstern
14	$4000	SCRGADGET	Screen-Gadget, wenn gesetzt
15	$8000	SYSGADGET	System-Gadget, wenn gesetzt

Wir wollen für unser Beispiel ein einfaches Boolean-Gadget und wählen dies durch:

```
        dc.w    1           ;Gadget-Typ: Boolean
```

Nun folgen einige Zeiger in der Gadget-Struktur. Der erste Zeiger enthält die Adresse der Image- oder Border-Struktur, welche für die Darstellung des Gadgets verwendet werden soll. Wird keine Darstellung benötigt, wird hier eine Null eingetragen. Wir wollen für das Gadget die Darstellung als Image wählen und setzen hier dafür den Zeiger auf die Image-Struktur ein, welche wir im Kapitel über Images erstellt haben:

```
        dc.l  image         ;Gadget-Image
```

Der nächste Zeiger wird nur verwendet, wenn das GADGHIMAGE-Flag im Flags-Wort der Struktur gesetzt ist. Dann nämlich wird dieser Zeiger auf eine weitere Struktur gerichtet, welche bei aktiviertem Gadget dargestellt werden soll. Wird jedoch eine Border-Struktur für die Gadget-Darstellung verwendet, so muß ggf. auch hier auf eine Border-Struktur gezeigt werden. Wir lassen die Darstellung eines zweiten Images erst einmal weg und setzen daher hier eine Null ein:

```
        dc.l    0           ;kein Select-Gadget
```

Der nächste Zeiger deutet auf eine Textstruktur des Textes, welcher bei dem Gadget ausgegeben werden soll. Wird kein Text benötigt, wird hier eine Null eingetragen. Wir wollen aber einen Text:

```
dc.l  ggtext        ;Gadget-Text
```

Nun folgt ein Langwort, welches bestimmt, welche Gadgets bei Aktivierung von diesem desaktiviert werden sollen. Diese Funktion ist allerdings noch nicht in Ordnung, so daß wir hier eine Null einsetzen können:

```
dc.l    0           ;kein Exclude
```

Auch den nächsten Zeiger können wir auf Null setzen, da er nur bei String- und Proportional-Gadgets verwendet wird. Bei einem von diesen muß er auf eine spezielle Struktur zur Beschreibung der Eigenschaften dieser Gadgets zeigen, SpecialInfo genannt.

```
dc.l    0           ;kein Special Info
```

Nun folgt ein Wort, in dem wir die laufende Nummer des Gadgets eintragen, welche als Gadget-Identifikation (ID) angesehen wird.

```
dc.w    1           ;Gadget-ID
```

Und schließlich noch ein Langwort, welches keine Funktion hat und daher auf Null gelegt werden kann:

```
dc.l    0           ;UserData (wird ignoriert)
```

Das war's. Hier noch einmal die Gadget-Struktur auf einen Blick:

```
gadget1:
     dc.l      0          ;kein weiteres Gadget
     dc.w     40          ;X-Position und
     dc.w     50          ;Y-Position des Gadgets
     dc.w     32          ;Breite und
     dc.w     13          ;Höhe der Hit-Box
     dc.w      4          ;Flags: Image, invertieren
     dc.w   $102          ;Activation-Flags
     dc.w      1          ;Gadget-Typ: Boolean
     dc.l  image          ;Gadget-Image
     dc.l      0          ;kein Select-Gadget
     dc.l ggtext          ;Gadget-Text
     dc.l      0          ;kein Exclude
     dc.l      0          ;kein Special Info
     dc.w      1          ;Gadget-ID
     dc.l      0          ;UserData (wird ignoriert)
```

Für das Image haben wir bereits eine Struktur vorliegen, die wir wiederverwenden können. Nun müssen wir noch den Text definieren, welcher unter dem Gadget stehen soll.

Da dieses Gadget wie ein Schalter aussieht, schlage ich die Beschriftung mit 'Switch' vor. Die Struktur dieses Textes wäre dann folgende:

```
ggtext:
     dc.b 1,0        ;Farben
     dc.b 1          ;Modus
even
     dc.w -8,14      ;X/Y-Position
     dc.l 0          ;Standard-Font
     dc.l swtext     ;Zeiger aud Text
     dc.l 0          ;kein weiterer Text

swtext:
     dc.b "Switch",0
even
```

Wenn Sie dies alles nun eingetippt haben, speichern Sie es zuerst einmal sicherheitshalber ab, assemblieren und starten es dann. Sie können nun den Schalter anklicken, was ihn sofort invertiert erscheinen läßt. Klicken Sie ihn ein zweites Mal an, wird er wieder in die normale Darstellung zurückgebracht.

Sie können jetzt an Hand der Tabellen einige Experimente mit diesen Strukturen anstellen. So können Sie durch Änderung des Flags von 4 auf 5 erreichen, daß das Gadget bei Aktivierung umrahmt wird. Oder versuchen Sie es mal mit gesetzten RELVERIFY-Bit (Bit 0: +1) in dem Activation-Flags-Wort. Sie können dann den Mauszeiger auf das Gadget stellen und die Taste drücken: es wird aktiviert. Halten Sie nun den Mausknopf gedrückt und bewegen die Maus, sehen Sie, daß beim Verlassen der Hit-Box die Aktivierung wieder aufgehoben wird. So wird erreicht, daß bei versehentlichem Anklicken des Gadgets nur mit gedrückter Taste aus dem Feld herausgefahren werden braucht, um die Aktivierung zu verhindern.

Und nun wollen wir den Schalter einmal umschalten. Dies ist recht einfach: wir brauchen lediglich eine weitere Image-Struktur für den ausgeschalteten Zustand. Dann setzen wir in die Gadget-Struktur den Zeiger auf diese Struktur hinter den Zeiger auf die normale Image-Struktur. Außerdem ändern wir das Flag-Wort auf 6, was die Darstellung dieses zweiten Images bei der Aktivierung bewirkt.

Hier die Image-Struktur für den eingeschalteten Schalter:

```
image2:
    dc.w 0,0        ;kein Offset
    dc.w 32,13      ;32x13 Bildpunkte
    dc.w 1          ;Modus 1
    dc.l imgdata2   ;Zeiger auf die Daten
    dc.b 2,1        ;gleiche Farben wie vorher
    dc.l 0          ;sonst nichts
```

```
imgdata2:          ;Daten für Schalter in ON-Stellung
 dc.l %00000000000000000000000000000000
 dc.l %00000000011100000000000000000000
 dc.l %00000000111110000011101001000000
 dc.l %00000000111110000010101101000000
 dc.l %00000000011110000010101011000000
 dc.l %00000000000011100011101001000000
 dc.l %00000000000001110000000000000000
 dc.l %00000000000000111000000000000000
 dc.l %00000000000011111111100000000000
 dc.l %00000000001111111111110000000000
 dc.l %00000000001111111111110000000000
 dc.l %00000000000011000000110000000000
 dc.l %00000000000000000000000000000000
```

Auf diese Weise wird sehr schön der Zustand des Gadgets angezeigt. Die 'Schalterstellung' stellt dabei den Zustand des Gadgets dar: ist dies aktiviert, steht der Schalter auf ON, andernfalls auf OFF.

So viel zu den Boolean-Gadgets. Was jetzt evtl. noch nicht erklärt wurde, können Sie wahrscheinlich experimentell herausbekommen. Gehen wir nun weiter zu den String-Gadgets, mit denen man auch feine Sachen machen kann.

7.9.2 String-Gadgets

Stellen Sie sich vor, Sie wollen in einem Programm Daten von der Diskette laden. Um den Benutzer dazu aufzufordern, den Dateinamen einzugeben, müßte ein entsprechender Text ausgegeben werden, der zur Eingabe des Namens auffordert, und dann eine Eingaberoutine aufgerufen werden, die die Tastatureingaben auswertet.

Einfacher und vor allem eleganter ist die Verwendung eines String-Gadgets. Diese Funktion erlaubt die komfortable Eingabe und/oder Änderung eines kurzen Textes mit wahlweiser Umrandung. Auch eine Undo-Funktion ist möglich, bei der durch Betätigung der rechten Amiga-Taste und 'Q' der alte Inhalt des Gadgets, also der alte Text wiederhergestellt wird.

Als weitere Besonderheit läßt sich die Größe des Textes und auch die des Eingabefeldes variieren. Ist der Text länger als das Eingabefeld breit, so wird der Text durch dieses Fensterchen hin- oder hergeschoben, wenn mit den Cursortasten oder bei normaler Eingabe an den Rand gelangt wird.

All dies sind Argumente für die String-Gadgets, was den Aufwand für Gadgets überhaupt durchaus rechtfertigt. Als zusätzliches Bonbon gibt es auch noch die Möglichkeit, die Eingabe auf Ziffern zu beschränken, um so die Eingabe einer Zahl zu ermöglichen. Die Umwandlung der Ziffernzeichen in eine binäre Zahl in einem Langwort übernimmt dann das Intuition, was auch eine große Erleichterung gerade für den Maschinensprache-Programmierer bedeutet! Ein so spezialisiertes String-Gadget nennt man dann Integer-Gadget.

Der Aufbau der Struktur ist weitgehend gleich derjenigen des Boolean-Gadgets. Es unterscheiden sich nur zwei Punkte erheblich:

Im Typ-Wort der Struktur muß nun eine 4 eingesetzt werden, um diesen Typ auf String-Gadget (STRGADGET) einzustellen.

Der Zeiger auf die SpecialInfo-Struktur wird diesmal benötigt. Wir setzen hier einen Zeiger ein, der auf die noch zu erstellende StringInfo-Struktur zeigt.

Die Angabe der Breite und Höhe des Gadgets in der Gadget-Struktur bekommt nun auch eine andere Bedeutung als vorher. Es wird zwar wieder der Bereich angegeben, in welchen man für die Aktivierung des String-Gadgets den Mauszeiger bringen und die Taste drücken muß, jedoch werden diese Angaben außerdem noch für die Darstellung des Textes maßgeblich. Dieser Wert stellt dann die Ausmaße des sogenannten Containers dar, in welchem der Text ausgegeben wird. Es empfiehlt sich, um diesen Container mit Hilfe der Border-Funktion einen Rahmen zu zeichnen, damit man dessen Ausmaße auch sieht.

Ist der Text länger als der Container, wird nur ein Teil dargestellt, welchen man durch Texteingaben oder Betätigungen der

links/rechts-Cursortasten durch den Container, den man als Fenster ansehen kann, durchschiebt. Die Zeichen, welche man eingibt, werden immer an der Cursorposition eingeschoben, d.h. der Rest des Textes wird um eins weitergeschoben. Um ein Zeichen zu ändern, muß man es vorher löschen. Die möglichen Funktionen für die Textbearbeitung sind folgende:

Cursortasten links/rechts

Bewegen des Cursors über den vorhandenen Text, ggf. durchschieben des Textes durch den Container

Cursortasten mit Shift

Setzen des Cursors an den Anfang bzw. ans Ende des Textes

DEL

Löschen des Zeichens unterm Cursor

BackSpace

Löschen des Zeichens links vom Cursor

Return

Abschließen der Texteingabe

Amiga rechts + Q

setzt den Text auf den ursprünglichen Inhalt zurück (Undo)

Die StringInfo-Struktur besitzt nur wenige Einträge:

Zuerst wird ein Zeiger angegeben, welcher auf den Speicherbereich zeigt, welcher für die Speicherung des eingegebenen Textes verwendet werden kann. Dieser Speicher-Puffer muß natürlich angemessen groß gewählt werden.

```
strinfo:
    dc.l    strpuffer   ;Zeiger auf Text-Puffer
```

Danach folgt ein Zeiger auf einen sogenannten Undo-Puffer. Dieser Zeiger und somit dieser Puffer werden nur benötigt, wenn die Undo-Funktion gewünscht ist. Dann muß der Puffer mindestens ebenso groß wie der Text-Puffer sein. Jedesmal bei Aufruf der String-Gadget-Funktion wird der Inhalt des Text-Puffers in diesen Puffer kopiert. Drücken Sie dann während der Eingabe die rechte Amiga-Taste und 'Q', so wird der alte Zustand des Textes wiederhergestellt, indem der Inhalt des Undo-Puffers wieder in den Text-Puffer kopiert wird. Verwendet man mehrere String-Gadgets in einem Programm, so kann man ein und denselben Undo-Puffer für alle einsetzen, da ja immer nur ein String-Gadget gleichzeitig bedient werden kann.

```
dc.l    undo        ;Zeiger auf Undo-Puffer
```

Das nun folgende Wort enthält die Cursorposition in dem Text. Man sollte dieses Wort auf Null setzen, damit beim Aufruf der Eingabe auch der ganze Text bzw. sein Anfang zu sehen ist.

```
dc.w    0           ;Cursor-Position
```

Als nächstes Wort wird die maximale Anzahl der einzugebenden Zeichen angegeben. Tippt man nach der Eingabe von so vielen Zeichen ein weiteres, so blinkt der Bildschirm kurz auf, um zu melden, daß es nicht weitergeht. Die Anzahl der Zeichen und der reservierte Platz für das Eingabefeld müssen nicht unbedingt übereinstimmen, da der Text ja beim Tippen durchgeschoben wird.

```
dc.w    10          ;max. Char
```

In das folgende Wort wird eingetragen, ab welchem Zeichen des Textes im Puffer der Text im Container ausgegeben werden soll. Auch hier sollte man eine Null einsetzen, damit man den Textanfang sieht.

```
dc.w    0           ;Text ab diesem Zeichen ausgeben
```

Nun folgen weitere 5 Worte, welche durch Intuition beschrieben werden und daher nicht initialisiert werden müssen. Es reicht daher aus, sie alle auf Null zu setzen. Diese Worte bekommen folgende Informationen:

```
        dc.w    0       ;Zeichenposition im Undo-Puffer
        dc.w    0       ;Anzahl der Zeichen im Text-Puffer
        dc.w    0       ;Anzahl der im Container sichtbaren
                         Zeichen
        dc.w    0       ;horizontaler Container-Offset
        dc.w    0       ;vertikaler Container-Offset
```

Danach folgen zwei Langworte, welche ebenfalls von Intuition initialisiert und beschrieben werden:

```
        dc.l    0       ;Zeiger auf aktuellen RastPort
        dc.l    0       ;Langwort mit Wert der Eingabe (bei
                         Integer-Gadgets)
```

Den Abschluß bildet ein Zeiger, welchen Sie auf eine Tastatur-Tabelle legen müssen, wenn sie das ALTKEYMAP-Flag des Gadgets gesetzt haben.

```
        dc.l    0       ;Standard-Tastaturbelegung
```

Hier noch einmal die StringInfo-Struktur auf einen Blick:

```
strinfo:
        dc.l    strpuffer   ;Zeiger auf Text-Puffer
        dc.l    undo        ;Zeiger auf Undo-Puffer
        dc.w    0           ;Cursor-Position
        dc.w    10          ;max. Char
        dc.w    0           ;Text ab diesem Zeichen ausgeben
        dc.w    0           ;Zeichenposition im Undo-Puffer
        dc.w    0           ;Anzahl der Zeichen im Text-Puffer
        dc.w    0           ;Anzahl der im Container sichtbaren
                            ;Zeichen
        dc.w    0           ;horizontaler Container-Offset
        dc.w    0           ;vertikaler Container-Offset
        dc.l    0           ;Zeiger auf aktuellen RastPort
        dc.l    0           ;Langwort mit Wert der Eingabe (bei
                             Integer-Gadgets)
        dc.l    0           ;Standard-Tastaturbelegung
```

Und nun noch den Text- und den Undo-Puffer:

```
strpuffer:
     dc.b    "Hallo !",0,0,0

undo:
     dc.l    0,0,0,0
even
```

Wenn Sie diese Zeilen eingegeben haben, können Sie entweder die alte Gadget-Struktur entsprechend ändern oder eine weitere aufbauen. Um den Schalter auch weiterhin im Bild zu haben und bedienen zu können, empfehle ich den Aufbau einer weiteren Gadget-Struktur. Ändern Sie dafür den ersten Zeiger der alten Struktur von 0 auf 'gadget1' und fügen dann die neue Struktur irgendwo ein. Hier eine Beispielstruktur für das String-Gadget mit den oben vorgegebenen Einträgen:

```
gadget1:                  ;* Struktur für String-Gadget
     dc.l    0            ;kein weiteres Gadget
     dc.w    20,80        ;Position
     dc.w    80,10        ;Breite/Höhe des Containers
     dc.w    0            ;Flags: normal
     dc.w    2            ;Activation ($802 für LongInt)
     dc.w    4            ;Typ: String-Gadget
     dc.l    border       ;Zeiger auf Umrahmung
     dc.l    0            ;keine Select-Zeichnung
     dc.l    0            ;kein Text
     dc.l    0            ;kein Exclude
     dc.l    strinfo      ;Zeiger auf StringInfo-Struktur
     dc.w    2            ;Gadget-ID
     dc.l    0            ;keine User-Daten

border:                   ;* Border für Container-Umrahmung
     dc.w    0,0          ;kein Offset
     dc.b    3,3          ;Farbe: rot
     dc.b    0            ;Modus: JAM1
     dc.b    5            ;5 XY-Paare
     dc.l    koord        ;Zeiger auf Koordinaten-Tabelle
     dc.l    0            ;keine Fortsetzung
```

```
koord:                  ;* Koordinaten für Umrahmung
     dc.w    -2,-2      ;Anfang links oben
     dc.w    80,-2      ;rechts oben
     dc.w    80,9       ;rechts unten
     dc.w    -2,9       ;links unten
     dc.w    -2,-2      ;zum Anfang
```

Mit den so vorgegebenen Daten erscheint im Fenster ein rotes Rechteck (der Border), in dem der Text 'Hallo !' steht. diesen Text können Sie ändern, indem Sie das Feld anklicken und nach dem Erscheinen des Cursors editieren. Haben Sie etwas falsches eingetippt, können Sie durch die Undo-Funktion (Amiga rechts und 'Q') wieder 'Hallo !' erscheinen lassen.

Wenn Sie nun etwas eingetippt haben und danach durch Return oder Klicken außerhalb des Feldes das Gadget desaktiviert haben (Cursor verschwindet), können Sie das Programm einmal abbrechen. Geben Sie dann im SEKA 'q strinfo' ein, so können Sie sich die Parameter sowie den Text und den Undo-Puffer ansehen.

Ändern Sie dann einmal das Activation-Flag auf $802 und den 'strpuffer' auf 'dc.l 0,0,0,0', assemblieren und starten dann das Programm. Sie können nun nach der Aktivierung des String-Gadgets wieder etwas eintippen, allerdings nur Ziffern. Die Eingabe eines Buchstabens wird durch ein Aufblinken des Bildschirms abgelehnt.

Geben Sie nun eine Zahl ein und beenden dann nach der Desaktivierung des Gadgets das Programm. Wenn Sie sich die StringInfo-Struktur nun mit 'q strinfo' ansehen, können sie den eben eingegeben Wert in sedezimaler Schreibweise im achten Langwort ('q strinfo+32') wiederfinden.

Kommen wir nun nach der Binär- und Text- bzw. Zahleneingabe zu den Gadgets, mit denen man einen analogen Wert durch einfaches Verschieben eines Symbols eingeben bzw. ändern kann: den Proportional-Gadgets.

7.9.3 Proportional-Gadgets

Sie kennen sicher die Vorzüge von Schiebereglern gegenüber Drehreglern, sei es in HiFi-Anlagen oder Toastern. Man kann bei ihnen den eingestellten Wert leicht erkennen, besonders wenn mehrere solche Regler nebeneinander liegen (z.B. Graphic-Equalizer in Verstärkern). Diese Schieberegler können wir auch auf dem Bildschirm des Amiga darstellen und mit der Maus bedienen, was uns sehr schöne Gestaltungsmöglichkeiten für unsere Programme bietet.

Realisiert wird dies wieder durch Gadgets. Bei diesem Gadget-Typ hat man die Möglichkeit, ein Symbol innerhalb eines Rahmens horizontal und/oder vertikal zu verschieben. Die Größe des Rahmens und des Schiebers kann variabel gestaltet werden. So kann die Rahmengröße relativ zur Fenstergröße sein und bei Veränderungen des Fensters ebenfalls variieren. Oder der Schieber kann so eingestellt werden, daß seine Größe innerhalb des Rahmens auch mit der möglichen Einstellbreite wächst bzw. schrumpft.

All diese Möglichkeiten (das waren nicht alle!) lassen sich am besten an Hand eines Beispiels erklären bzw. ausprobieren. Dafür wollen wir einmal ein einfaches Proportional-Gadget erstellen, welches horizontal verschoben werden kann.

Wir brauchen wieder eine Gadget-Struktur, welche den gleichen Aufbau wie die anderen hat. Um die Änderungen zu verdeutlichen, gebe ich Ihnen hier eine komplette Beispielstruktur für unser Gadget. Sie können dieses Gadget zusätzlich zu den anderen darstellen lassen, wenn Sie in dem ersten Langwort der letzten Struktur 'dc.l gadget2' eintragen.

```
gadget2:            ;* Struktur für ein Proportional-Gadget
     dc.l    0           ;kein weiteres Gadget
     dc.w    150,30      ;Position
     dc.w    100,10      ;Breite und Höhe des Rahmens
     dc.w    4           ;Flags: GADGIMAGE
     dc.w    2           ;Activation: GADGIMMEDIATE
     dc.w    3           ;Typ: Proportional-Gadget
     dc.l    mover       ;Zeiger auf Regler-Daten
     dc.l    0           ;keine Select-Struktur
     dc.l    0           ;kein Text
     dc.l    0           ;kein Exclude
     dc.l    propinfo    ;Zeiger auf PropInfo-Struktur
     dc.w    3           ;Gadget-ID
     dc.l    0           ;keine User-Daten
```

Sie sehen hier zwei Besonderheiten: es wird auf eine Image-Struktur gezeigt, welche für den Schieber verwendet wird, und als SpecialInfo-Zeiger ein Zeiger auf eine weitere Struktur eingetragen.

Zuerst die Struktur 'mover', welche eine Image-Struktur für den Schieber darstellt. Hier ein Beispiel für diese Struktur:

```
mover:                  ;* Struktur für Schieber-Image
     dc.w    0,0         ;kein Offset
     dc.w    16,7        ;16x7 Pixel groß
     dc.w    1           ;eine Bit-Plane
     dc.l    moverdata   ;Zeiger auf Image-Daten
     dc.b    1,0         ;Farbe: weiß
     dc.l    0           ;keine Fortsetzung

moverdata:              ;* Image-Daten für Schieber
 dc.w %0111111111111110
 dc.w %0101111111111010
 dc.w %0101011111101010
 dc.w %0101010110101010
 dc.w %0101011111101010
 dc.w %0101111111111010
 dc.w %0111111111111110
```

Bisher war nichts neues dabei. Nun aber zu der PropInfo-Struktur, welche die Eigenarten des Proportional-Gadgets beschreibt:

Die Struktur beginnt mit einem Flags-Wort, welches folgende Flag-Bits beinhaltet:

Bit	Wert	Name	Bedeutung: Nachricht bei
0	1	AUTOKNOB	Schieber wird automatisch erstellt
1	2	FREEHORIZ	erlaubt horizontale Verschiebung
2	4	FREEVERT	erlaubt vertikale Verschiebung
3	8	PROPBORDERLESS	schaltet automatische Umrandung ab
8	$100	KNOBHIT	wird gesetzt, wenn der Schieber berührt wird

Die ersten vier Bits können Sie setzen, um die von Ihnen gewünschte Darstellung einzustellen. Bit 8 wird von Intuition gesetzt, wenn der Schieber mit dem Mauszeiger angeklickt wird.

Bit 0, AUTOKNOB, erlaubt die einfachste Art des Proportional-Gadgets. Wird dieses Bit gesetzt, so werden keine Daten für das Schieber-Image mehr angenommen. Stattdessen wird ein weißer Schieber generiert, welcher in den Ausmaßen an die Box und den dargestellten Wert angepaßt wird. Diese Anpassung als Beispiel:

Wird bei einem Textprogramm eine vertikale Verschiebung zugelassen und mit diesem Schieber die angezeigten Zeilen eines langen Textes eingestellt, so ist ja ein bestimmter Prozentsatz des Textes in dem Textfenster sichtbar. Das Verhältnis zwischen der gesamten Zeilenzahl und den dargestellten Zeichen drückt sich bei einem AUTOKNOB in dem Verhältnis zwischen der Umrandung und dem Schieber aus. Je höher dieser Prozentsatz, desto größer wird auch der Schieber in der Umrandung.

Wir wollen diese Möglichkeit einmal außer acht lassen, obwohl sie einfach und interessant ist, da ein einfacher weißer Knopf nicht besonders schön ist. Wenn Sie es dennoch ausprobieren wollen, so achten Sie darauf, daß Sie den Zeiger auf die Imagedaten nicht auf Null, sondern auf einen 4 Worte langen Puffer stellen, in den Intuition dann folgende Werte einträgt. Dieser Puffer kann dann so aussehen:

```
puffer:
        dc.w    0       ;X-Position des Schiebers in der Box
        dc.w    0       ;Y-Position in der Box
        dc.w    0       ;Breite des Schiebers
        dc.w    0       ;Höhe des Schiebers
```

Zurück zu der PropInfo-Struktur. Da wir den AUTOKNOB nicht verwenden und nur horizontale Verschiebung zulassen wollen, setzen wir als Flag eine 2 ein:

```
propinfo:
        dc.w    2       ;Flags: FREEHORIZ
```

In den nächsten beiden Worten der Struktur werden die horizontale (HorizPot) und vertikale (VertPot) Position des Schiebers eingetragen. Hierbei bedeutet der Wert 0 links bzw. oben und der Wert $FFFF rechts bzw. unten. Der mit Hilfe der Maus eingestellte Wert nach der Verschiebung liegt dann irgendwo dazwischen. Wir setzen diese Werte für den Anfang auf 0, nach einer Verschiebung finden Sie dann hier den neuen Wert.

```
        dc.w    0,0     ;X/Y-Position des Schiebers
```

Nun folgen zwei weitere Worte, welche die Größe des AUTOKNOBs oder die Schrittweite des Schiebers angeben, mit welcher er sich bewegt, wenn Sie in der Box neben den Schieber klicken. Diese Worte heißen HorizBody für die horizontale und VertBody für die vertikale Verschiebung.

Wenn Sie, wie z.B. bei der Einstellung der Farbmischungen im Preferences-Programm, insgesamt 16 Stufen bei der Verschiebung zulassen wollen, so stellen Sie diesen Wert auf die maximale Position/16, also $FFFF/$10=$FFF. Durch Anklicken der Box neben dem Schieber bewegt dieser sich dann um 1/16 der gesamten Breite der Box. Bei Verwendung der AUTOKNOB-Option ist dieser Knopf auch 1/16 so breit wie die Box. Wir stellen dieses Beispiel auch einmal ein:

```
        dc.w    $ffff/16        ;horizontale Schrittweite: 1/16
        dc.w    0               ;keine vertikale Verschiebung
```

Nun folgen wieder 6 Worte, die von Intuition initialisiert und beschrieben werden:

```
        dc.w    0           ;Box-Breite
        dc.w    0           ;Box-Höhe
        dc.w    0           ;absolute Schrittweite horizontal
        dc.w    0           ;und vertikal
        dc.w    0           ;linker Rand der Box
        dc.w    0           ;oberer Rand der Box
```

Das war's. Hier noch einmal die PropInfo-Struktur auf einen Blick:

```
propinfo:
        dc.w    2           ;Flags: FREEHORIZ
        dc.w    0,0         ;X/Y-Position des Schiebers
        dc.w    $ffff/16    ;horizontale Schrittweite: 1/16
        dc.w    0           ;keine vertikale Verschiebung
        dc.w    0           ;Box-Breite
        dc.w    0           ;Box-Höhe
        dc.w    0           ;absolute Schrittweite horizontal
        dc.w    0           ;und vertikal
        dc.w    0           ;linker Rand der Box
        dc.w    0           ;oberer Rand der Box
```

Wenn Sie dies eingegeben haben, können Sie das Programm ja einmal starten und ausprobieren. Die eingestellte Position können Sie sich dann nach Abbruch des Programms durch 'q propinfo' ansehen.

Danach können Sie auch ausprobieren, die vertikale Verschiebung zuzulassen, indem Sie das Flags-Wort auf 6 setzen, die vertikale Schrittweite auf $FFFF/10 und die Höhe des Gadgets z.B. auf 80 stellen. Auch die Verwendung des AUTOKNOBs sollten Sie einmal durch Einsetzen einer 7 in die Flags probieren.

8. Beispielprogramme

Sie haben nun eine ganze Menge kennengelernt, was Sie mit der Maschinensprache im Amiga bewirken können. Was nun noch fehlt, sind einige grundlegenden Dinge, die Sie als Handwerkzeug für die Programmierung brauchen. Diesen Mangel wollen wir nun beseitigen.

Wir werden nun einige Beispielprogramme betrachten, die manchen im Buch erwähnten Mythos aufdeckt. Diese Programme sind dann leicht für Ihre eigenen Programme verwendbar, so daß dafür so gut wie keine Grenzen mehr bestehen.

8.1 Umschalten in den Supervisor-Modus

Wie bereits im Kapitel über den MC68000-Prozessor erwähnt, besitzt dieser zwei verschiedene Betriebszustände: den User- und den Supervisor-Modus. Es ist oft nötig, zwischen diesen Modi hin- und herzuschalten, was allerdings nicht ganz so einfach ist.

Das Problem bei der Sache ist das, daß man im User-Modus nicht auf das Status-Register zugreifen darf. Ein Schreibzugriff hierauf hat dann eine Exception zur Folge (Privilege Violation = Privilegs-Verletzung), was schließlich mit der ach so beliebten Guru-Meditation endet.

Wie ist es nun möglich, überhaupt in den Supervisor-Modus zu gelangen?

Kein Problem. Das Betriebssystem des Amiga bzw. die EXEC-Bibliothek beinhaltet eine Funktion, die dies für uns bewerkstelligt. Diese Funktion heißt SuperState und benötigt keine Parameter. Wir können diese Funktion leicht aufrufen, und zwar mit folgenden Zeilen:

```
ExecBase   = 4       ;EXEC-Basisadresse
SuperState = -150    ;Umschalt-Funktion

     ...
     move.l  ExecBase,a6       ;EXEC-Basisadresse in A6
     jsr     SuperState(a6)    ;umschalten in Supervisor
     move.l  d0,savesp         ;Rückgabewert retten

     ...
savesp:  blk.l 1               ;Platz für SP-Wert
```

Wir erhalten im Register D0 den Stand des Stack-Pointers (SP) zurück, welcher beim Aufruf gesetzt wurde. Dies finden Sie zwar auch in dem Register A7, jedoch verändert sich dieses Register ständig. Der Grund hierfür ist der, daß im Supervisor-Modus und damit mit dem SP alle Interrupts des Amiga arbeiten, die ja gerade bei diesem Rechner ziemlich zahlreich sind. Auf diese Interrupts kommen wir gleich noch zurück.

Verwendet wird nach diesem Aufruf weiterhin der User-Stack und nicht der Supervisor-Stack. Dadurch haben Sie weiterhin die Möglichkeit, auf den alten User-Stack zuzugreifen. Zu beachten ist allerdings, daß der User-Stack groß genug ist, damit die erwähnten Interrupts auch genug Platz für ihre Daten auf dem Stack haben.

Was wir im Register D0 geliefert bekommen haben, muß zunächst einmal gerettet werden, da wir diesen Wert später noch brauchen. Schließlich müssen wir irgendwann auch wieder in den User-Modus zurückkommen, wofür wir eine weitere Funktion des EXEC zur Verfügung haben. Es handelt sich dabei um die UserState-Funktion, welche einen Parameter braucht: den von der SuperState-Funktion gelieferten SP-Wert.

Da wir diesen Wert in dem Langwort ab 'savesp' gerettet haben, können wir nun schreiben:

```
UserState  = -156

        ...
        move.l  ExecBase,a6       ;EXEC-Basisadresse in A6
        move.l  savesp,d0         ;alten SP in D0 übergeben
        jsr     UserState(a6)     ;zurück in User-Modus
```

Wir sind nun wieder im User-Modus, und zwar mit dem selben Stand des User-Stack-Pointers (USP) wie vorher. Aus diesem Grund können Sie auch eine Funktion, welche im Supervisor-Modus ausgeführt werden soll, einfach als Unterroutine schreiben. Diese beginnt dann mit dem Aufruf von SuperState, rettet den Rückgabewert, führt die gewünschte Funktion aus, ruft UserState auf und endet mit RTS. Wäre der Stand des USP an dieser Stelle abweichend vom Anfangszustand, so würde dieser RTS-Befehl natürlich nicht richtig funktionieren und der Rechner würde 'ins Blaue springen' und wahrscheinlich abstürzen. Hier klappt es aber.

Nun taucht die Frage auf: Wie bringt das Betriebssystem das fertig, in den Supervisor-Modus umzuschalten? Dies ist recht einfach zu erklären, und zwar läuft dies etwa so ab:

In der SuperState-Funktion wird einfach ein Zugriff auf das Status-Register versucht. Der Effekt ist der, daß dadurch eine Exception auftritt und eine Routine aufgerufen wird, deren Adresse im Langwort ab $20 steht. Es handelt sich dabei um den Exception-Vektor für die Privilegs-Verletzung. Die Routine, in die also verzweigt wird, wird automatisch im Supervisor-Modus aufgerufen. Dort wird nun getestet, von wo aus diese Exception ausgelöst wurde.

Findet die Routine heraus, daß der Ursprung dieser Exception in der SuperState-Routine der EXEC-Bibliothek, deren Adresse ihr ja bekannt ist, liegt, so ist die Sache für sie erledigt. Es wird dann einfach in die Routine zurückverzweigt, ohne wieder in den User-Modus umzuschalten. Das war schon alles.

8.2 Exception-Programmierung

Die bereits im Prozessor-Kapitel beschriebenen Exceptions bieten eine große Zahl von Möglichkeiten, die Funktionen des Amiga in eigene Bahnen zu lenken. Wir können Fehler im System in eigene Bahnen lenken und so vielleicht sogar den Guru überlisten.

Hier noch einmal die Liste der Vektoren, welche für den Sprung in die entsprechenden Exception-Routinen verwendet werden:

Nummer	Adresse	Verwendung bei
2	*$008*	Bus Error
3	*$00C*	Address Error
4	*$010*	Illegaler Befehl
5	*$014*	Division durch Null
6	*$018*	CHK-Befehl
7	*$01C*	TRAPV-Befehl
8	*$020*	Privilegsverletzung
9	*$024*	Trace
10	*$028*	Axxx-Befehlsemulation
11	*$02C*	Fxxx-Befehlsemulation
	$030-$038	Reserviert
15	*$03C*	Uninitialisierter Interrupt
	$040-$05F	Reserviert
24	*$060*	Unberechtigter Interrupt
25-31	*$064-$083*	Level 1-7 Interrupt
32-47	*$080-$0BF*	TRAP-Befehle
	$0C0-$0FF	Reserviert
64-255	*$100-$3FF*	User-Interrupt-Vektoren

Nehmen wir nun als Beispiel die TRAP-Befehle, welche im Amiga-Betriebssystem nicht verwendet werden. Ein TRAP-Befehl wird in einem Maschinenprogramm zusammen mit einer Nummer zwischen 0 und 15 eingesetzt, um eine der 16 möglichen TRAP-Routinen aufzurufen. Wird der Befehl TRAP #0 ausgeführt, so verzweigt der Prozessor (im Supervisor-Modus) in

die Routine, deren Adresse ab $80 im Speicher liegt. Diese Routine muß mit einem RTE-Befehl (Return from Exception) enden.

Einige Betriebssysteme, z.B. das TOS des ATARI ST, sind vollständig über diese TRAPs aufzurufen. Parameter werden dabei auf dem Stack übergeben und dann je nach gewünschten Teil des Betriebssystems ein TRAP-Befehl ausgeführt. Der Vorteil dessen ist, daß man keine Adressen aus dem Betriebssystem kennen muß, wie dies ja beim Amiga der Fall ist (ExecBase = 4).

Wir wollen nun eine eigene TRAP-Routine schreiben, welche die Anwendung der TRAPs demonstriert. Dazu brauchen wir 3 Programmteile:

1. die Initialisierung des TRAP-Vektors
2. die TRAP-Routine selbst, die mit RTE enden muß
3. eine Test-Routine, in der der TRAP-Befehl aufgerufen wird

Die Initialisierung ist sehr einfach:

```
init:
    move.l  #trap0,$80        ;Vektor für TRAP #0 setzen
    rts
```

Die Routine mit dem Namen 'trap0' muß nun noch erstellt werden. Nehmen wir dafür als Beispiel das Programm aus dem Hardware-Kapitel, welches einen Piepton erzeugt.

Wir schreiben diese Routine nur geringfügig um, und zwar setzen wir anstelle von RTS ein RTE ans Ende, löschen die Zeile, in der dem Schleifenzähler D0 für die Ton-Verzögerung geladen wird und ändern schließlich die Warteschleife dahingehend, daß sie auch mit Langworten arbeitet. Somit können wir durch das Laden dieses Registers mit einem beliebigen Wert und nachfolgenden TRAP #0 einen Piepton beliebiger Dauer auslösen.

```
;** Piepton-Erzeugung bei TRAP #0 **

ctlw  = $dff096                ;DMA-Control

c0thi = $dff0a0                ;Tabellenadresse HI
c0tlo = c0thi+2                ;Tabellenadresse LO
c0tl  = c0thi+4                ;Tabellenlänge
c0per = c0thi+6                ;Abtastrate
c0vol = c0thi+8                ;Lautstärke

trap0:                         ;* kurzen Piep-Ton erzeugen
     move.l  #table,c0thi      ;Tabellenanfang
     move    #4,c0tl           ;Tabellenlänge
     move    #300,c0per        ;Abtastrate
     move    #40,c0vol         ;Lautstärke

     move    #$8201,ctlw       ;Start DMA bzw. Sound

loop:
     subq.l  #1,d0             ;Zähler -1
     bne     loop              ;bis 0 herunterzählen

still:
     move #1,ctlw              ;und Ton abschalten
     rte                       ;Exception-Ende

table:                         ;Sound-Tabelle
     dc.b -40,-70,-40,0,40,70,40,0
```

Sie müssen hier wiederum beachten, daß die Tabelle 'table' im Chip-RAM ($00000-$7FFFF) liegt, da sonst der Sound-Chip nicht an diese Daten herankommen kann!

Wenn Sie dies eingegeben haben, können Sie nun noch zum Testen folgende kleine Routine anfügen:

```
test:
     move.l  #$2ffff,d0        ;Tondauer in D0 übergeben
     trap    #0                ;Exception auslösen: Piep
     rts
```

Nun assemblieren Sie dies alles mit 'a' und zweimal Return (SEKA) und starten anschließend die Initialisierungs-Routine mit 'j init'. Noch passiert nichts.

Nun geben Sie 'j test' ein: es ertönt der Piep-Ton von etwa einer Sekunde Dauer!

Eines ist dabei unbedingt zu beachten: Wenn Sie irgendetwas in dem Programm ändern und neu assemblieren, so kann sich ja die Adresse der 'trap0'-Routine ändern. Bevor Sie dann wieder den TRAP-Befehl ausführen lassen, müssen Sie unbedingt die Initialisierung wiederholen, damit der Rechner nicht an die falsche Stelle verzweigt!

Anhang

Bibliotheksfunktionen im Überblick

Die nun folgende Tabelle gibt Ihnen einen Überblick über alle verfügbaren Bibliotheken mit ihren Funktionen. Die Tabelle beginnt jeweils als Überschrift mit dem Namen der Bibliothek, in der die folgenden Funktionen liegen.

Diese Funktionen werden jeweils mit ihrem negativen Offset hex und dezimal, ihrem Namen und schließlich ihren Übergabeparameter dargestellt. Die Parameternamen stehen in Klammern hinter dem Funktionsnamen, die zweite Klammer enthält in gleicher Reihenfolge die Register, in denen diese Parameter übergeben werden müssen. Werden keine Parameter benötigt, wird dies durch () angedeutet.

clist.library

```
-$001E   -30   InitCLPool (cLPool, size)(A0,D0)
-$0024   -36   AllocCList (cLPool)(A1)
-$002A   -42   FreeCList (cList)(A0)
-$0030   -48   FlushCList (cList)(A0)
-$0036   -54   SizeCList (cList)(A0)
-$003C   -60   PutCLChar (cList,byte)(A0,D0)
-$0042   -66   GetCLChar (cList)(A0)
-$0048   -72   UnGetCLChar (cList,byte)(A0,D0)
-$004E   -78   UnPutCLChar (cList)(A0)
-$0054   -84   PutCLWord (cList,word)(A0,D0)
-$005A   -90   GetCLWord (cList)(A0)
-$0060   -96   UnGetCLWord (cList,word)(A0,D0)
-$0066  -102   UnPutCLWord (cList)(A0)
-$006C  -108   PutCLBuf (cList,buffer,length)(A0,A1,D1)
-$0072  -114   GetCLBuf (cList,buffer,maxLength)(A0,A1,D1)
-$0078  -120   MarkCList (cList,offset)(A0,D0)
-$007E  -126   IncrCLMark (cList)(A0)
-$0084  -132   PeekCLMark (cList)(A0)
-$008A  -138   SplitCList (cList)(A0)
-$0090  -144   CopyCList (cList)(A0)
-$0096  -150   SubCList (cList,index,length)(A0,D0,D1)
-$009C  -156   ConcatCList (sourceCList,destCList)(A0,A1)
```

console.library

```
-$002A   -42   CDInputHandler (events,device)(A0,A1)
-$0030   -48   RawKeyConvert (events,buffer,length,keyMap) (A0,A1,D1,A2)
```

diskfont.library

```
-$001E   -30   OpenDiskFont (textAttr)(A0)
-$0024   -36   AvailFonts (buffer,bufBytes,flags)(A0,D0,D1)
```

dos.library

```
-$001E   -30   Open (name,accessMode)(D1,D2)
-$0024   -36   Close (file)(D1)
-$002A   -42   Read (file,buffer,length)(D1,D2,D3)
-$0030   -48   Write (file,buffer,length)(D1,D2,D3)
-$0036   -54   Input ()
-$003C   -60   Output ()
-$0042   -66   Seek (file,position,offset)(D1,D2,D3)
-$0048   -72   DeleteFile (name)(D1)
-$004E   -78   Rename (oldName,newName)(D1,D2)
-$0054   -84   Lock (name,type)(D1,D2)
-$005A   -90   UnLock (lock)(D1)
-$0060   -96   DupLock (lock)(D1)
-$0066  -102   Examine (lock,fileInfoBlock)(D1,D2)
-$006C  -108   ExNext (lock,fileInfoBlock)(D1,D2)
-$0072  -114   Info (lock,parameterBlock)(D1,D2)
-$0078  -120   CreateDir (name)(D1)
-$007E  -126   CurrentDir (lock)(D1)
-$0084  -132   IoErr ()
-$008A  -138   CreateProc (name,pri,segList,stackSize)(D1,D2,D3,D4)
-$0090  -144   Exit (returnCode)(D1)
-$0096  -150   LoadSeg (fileName)(D1)
-$009C  -156   UnLoadSeg (segment)(D1)
-$00A2  -162   GetPacket (wait)(D1)
-$00A8  -168   QueuePacket (packet)(D1)
-$00AE  -174   DeviceProc (name)(D1)
-$00B4  -180   SetComment (name,comment)(D1,D2)
-$00BA  -186   SetProtection (name,mask)(D1,D2)
-$00C0  -192   DateStamp (date)(D1)
-$00C6  -198   Delay (timeout)(D1)
```

```
-$00CC  -204  WaitForChar (file,timeout)(D1,D2)
-$00D2  -210  ParentDir (lock)(D1)
-$00D8  -216  IsInteractive (file)(D1)
-$00DE  -222  Execute (string,file,file)(D1,D2,D3)
```

exec.library

```
-$001E   -30   Supervisor ()
-$0024   -36   ExitIntr ()
-$002A   -42   Schedule ()
-$0030   -48   Reschedule ()
-$0036   -54   Switch ()
-$003C   -60   Dispatch ()
-$0042   -66   Exception ()
-$0048   -72   InitCode (startClass,version)(D0,D1)
-$004E   -78   InitStruct (initTable,memory,size)(A1,A2,D0)
-$0054   -84   MakeLibrary (funcInit,structInit,libInit,dataSize,
               codeSize) (A0,A1,A2,D0,D1)
-$005A   -90   MakeFunctions (target,functionArray,funcDispBase)
               (A0,A1,A2)
-$0060   -96   FindResident (name)(A1)
-$0066  -102   InitResident (resident,segList)(A1,D1)
-$006C  -108   Alert (alertNum,parameters)(D7,A5)
-$0072  -114   Debug ()
-$0078  -120   Disable ()
-$007E  -126   Enable ()
-$0084  -132   Forbid ()
-$008A  -138   Permit ()
-$0090  -144   SetSR (newSR,mask)(D0,D1)
-$0096  -150   SuperState ()
-$009C  -156   UserState (sysStack)(D0)
-$00A2  -162   SetIntVector (intNumber,interrupt)(D0,A1)
-$00A8  -168   AddIntServer (intNumber,interrupt)(D0,A1)
-$00AE  -174   RemIntServer (intNumber,interrupt)(D0,A1)
-$00B4  -180   Cause (interrupt)(A1)
-$00BA  -186   Allocate (freeList,byteSize)(A0,D0)
-$00C0  -192   Deallocate (freeList,memoryBlock,byteSize)(A0,A1,D0)
-$00C6  -198   AllocMem (byteSize,requirements)(D0,D1)
-$00CC  -204   AllocAbs (byteSize,location)(D0,A1)
-$00D2  -210   FreeMem (memoryBlock,byteSize)(A1,D0)
-$00D8  -216   AvailMem (requirements)(D1)
-$00DE  -222   AllocEntry (entry)(A0)
-$00E4  -228   FreeEntry (entry)(A0)
-$00EA  -234   Insert (list,node,pred)(A0,A1,A2)
-$00F0  -240   AddHead (list,node)(A0,A1)
```

```
-$00F6  -246  AddTail (list,node)(A0,A1)
-$00FC  -252  Remove (node)(A1)
-$0102  -258  RemHead (list)(A0)
-$0108  -264  RemTail (list)(A0)
-$010E  -270  Enqueue (list,node)(A0,A1)
-$0114  -276  FindName (list,name)(A0,A1)
-$011A  -282  AddTask (task,initPC,finalPC)(A1,A2,A3)
-$0120  -288  RemTask (task)(A1)
-$0126  -294  FindTask (name)(A1)
-$012C  -300  SetTaskPri (task,priority)(A1,D0)
-$0132  -306  SetSignal (newSignals,signalSet)(D0,D1)
-$0138  -312  SetExcept (newSignals,signalSet)(D0,D1)
-$013E  -318  Wait (signalSet)(D0)
-$0144  -324  Signal (task,signalSet)(A1,D0)
-$014A  -330  AllocSignal (signalNum)(D0)
-$0150  -336  FreeSignal (signalNum)(D0)
-$0156  -342  AllocTrap (trapNum)(D0)
-$015C  -348  FreeTrap (trapNum)(D0)
-$0162  -354  AddPort (port)(A1)
-$0168  -360  RemPort (port)(A1)
-$016E  -366  PutMsg (port,message)(A0,A1)
-$0174  -372  GetMsg (port)(A0)
-$017A  -378  ReplyMsg (message)(A1)
-$0180  -384  WaitPort (port)(A0)
-$0186  -390  FindPort (name)(A1)
-$018C  -396  AddLibrary (library)(A1)
-$0192  -402  RemLibrary (library)(A1)
-$0198  -408  OldOpenLibrary (libName)(A1)
-$019E  -414  CloseLibrary (library)(A1)
-$01A4  -420  SetFunction (library,funcOffset,funcEntry)(A1,A0,D0)
-$01AA  -426  SumLibrary (library)(A1)
-$01B0  -432  AddDevice (device)(A1)
-$01B6  -438  RemDevice (device)(A1)
-$01BC  -444  OpenDevice (devName,unit,ioRequest,flags)(A0,D0,A1,D1)
-$01C2  -450  CloseDevice (ioRequest)(A1)
-$01C8  -456  DoIO (ioRequest)(A1)
-$01CE  -462  SendIO (ioRequest)(A1)
-$01D4  -468  CheckIO (ioRequest)(A1)
-$01DA  -474  WaitIO (ioRequest)(A1)
-$01E0  -480  AbortIO (ioRequest)(A1)
-$01E6  -486  AddRescource (rescource)(A1)
-$01EC  -492  RemRescource (rescource)(A1)
-$01F2  -498  OpenRescource (resName,version)(A1,D0)
-$01F8  -504  RawIOInit ()
-$01FE  -510  RawMayGetChar ()
-$0204  -516  RawPutChar (char)(D0)
```

```
-$020A  -522   RawDoFmt ()(A0,A1,A2,A3)
-$0210  -528   GetCC ()
-$0216  -534   TypeOfMem (address)(A1)
-$021C  -540   Procedure (semaport,bidMsg)(A0,A1)
-$0222  -546   Vacate (semaport)(A0)
-$0228  -552   OpenLibrary (libName,version)(A1,D0)
```

graphics.library

```
-$001E   -30   BltBitMap (srcBitMap,srcX,srcY,destBitMap,destX,destY,
               sizeX,sizeY,minterm,mask,tempA) (A0,D0,D1,A1,D2,D3,D4,
               D5,D6,D7,A2)
-$0024   -36   BltTemplate (source,srcX,srcMod,destRastPort,destX,destY,
               sizeX,sizeY)(A0,D0,D1,A1,D2,D3,D4,D5)
-$002A   -42   ClearEOL (rastPort)(A1)
-$0030   -48   ClearScreen (rastPort)(A1)
-$0036   -54   TextLength (RastPort,string,count)(A1,A0,D0)
-$003C   -60   Text (RastPort,String,count)(A1,A0,D0)
-$0042   -66   SetFont (RAstPortID,textFont)(A1,A0)
-$0048   -72   OpenFont (textAttr)(A0)
-$004E   -78   CloseFont (textFont)(A1)
-$0054   -84   AskSoftStyle (rastPort)(A1)
-$005A   -90   SetSoftStyle (rastPort,style,enable)(A1,D0,D1)
-$0060   -96   AddBob (bob,rastPort)(A0,A1)
-$0066  -102   AddVSprite (vSprite,rastPort)(A0,A1)
-$006C  -108   DoCollision (rastPort)(A1)
-$0072  -114   DrawGList (rastPort,viewPort)(A1,A0)
-$0078  -120   InitGels (dummyHead,dummyTail,GelsInfo)(A0,A1,A2)
-$007E  -126   InitMasks (vSprite)(A0)
-$0084  -132   RemIBob (bob,rastPort,viewPort)(A0,A1,A2)
-$008A  -138   RemVSprite (vSprite)(A0)
-$0090  -144   SetCollision (type,routine,gelsInfo)(D0,A0,A1)
-$0096  -150   SortGList (rastPort)(A1)
-$009C  -156   AddAnimObj (obj,animationKey,rastPort)(A0,A1,A2)
-$00A2  -162   Animate (animationKey,rastPort)(A0,A1)
-$00A8  -168   GetGBuffers (animationObj,rastPort,doubleBuffer)
               (A0,A1,D0)
-$00AE  -174   InitGMasks (animationObj)(A0)
-$00B4  -180   GelsFuncE ()
-$00BA  -186   GelsFuncF ()
-$00C0  -192   LoadRGB4 (viewPort,colors,count)(A0,A1,D0)
-$00C6  -198   InitRastPort (rastPort)(A1)
-$00CC  -204   InitVPort (viewPort)(A0)
-$00D2  -210   MrgCop (view)(A1)
-$00D8  -216   MakeVPort (view,viewPort)(A0,A1)
```

```
-$00DE  -222   LoadView (view)(A1)
-$00E4  -228   WaitBlit ()
-$00EA  -234   SetRast (rastPort,color)(A1,D0)
-$00F0  -240   Move (rastPort,x,y)(A1,D0,D1)
-$00F6  -246   Draw (rastPort,x,y)(A1,D0,D1)
-$00FC  -252   AreaMove (rastPort,x,y)(A1,D0,D1)
-$0102  -258   AreaDraw (rastPort,x,y)(A1,D0,D1)
-$0108  -264   AreaEnd (rastPort)(A1)
-$010E  -270   WaitTOF ()
-$0114  -276   QBlit (blit)(A1)
-$011A  -282   InitArea (areaInfo,vectorTable,vectorTableSize)(A0,A1,D0)
-$0120  -288   SetRGB4 (viewPort,index,r,g,b)(A0,D0,D1,D2,D3)
-$0126  -294   QBSBlit (blit)(A1)
-$012C  -300   BltClear (memory,size,flags)(A1,D0,D1)
-$0132  -306   RectFill (rastPort,xl,yl,xu,yu)(A1,D0,D1,D2,D3)
-$0138  -312   BltPattern (rastPort,ras,xl,yl,maxX,maxY,fillBytes)
               (A1,A0,D0,D1,D2,D3,D4)
-$013E  -318   ReadPixel (rastPort,x,y)(A1,D0,D1)
-$0144  -324   WritePixel (rastPort,x,y)(A1,D0,D1)
-$014A  -330   Flood (rastPort,mode,x,y)(A1,D2,D0,D1)
-$0150  -336   PolyDraw (rastPort,count,polyTable)(A1,D0,A0)
-$0156  -342   SetAPen (rastPort,pen)(A1,D0)
-$015C  -348   SetBPen (rastPort,pen)(A1,D0)
-$0162  -354   SetDrMd (rastPort,drawMode)(A1,D0)
-$0168  -360   InitView (view)(A1)
-$016E  -366   CBump (copperList)(A1)
-$0174  -372   CMove (copperList,destination,data)(A1,D0,D1)
-$017A  -378   CWait (copperList,x,y)(A1,D0,D1)
-$0180  -384   VBeamPos ()
-$0186  -390   InitBitMap (bitMap,depth,width,heigth)(A0,D0,D1,D2)
-$018C  -396   ScrollRaster (rastPort,dX,dY,minx,miny,maxx,maxy)(A1,D0,
               D1,D2,D3,D4,D5)
-$0192  -402   WaitBOVP (viewPort)(A0)
-$0198  -408   GetSprite (simpleSprite,num)(A0,D0)
-$019E  -414   FreeSprite (num)(D0)
-$01A4  -420   ChangeSprite (vp,simpleSprite,data)(A0,A1,A2)
-$01AA  -426   MoveSprite (viewPort,simpleSprite,x,y)(A0,A1,D0,D1)
-$01B0  -432   LockLayerRom (layer)(A5)
-$01B6  -438   UnlockLayerRom (layer)(A5)
-$01BC  -444   SyncSBitMap (1)(A0)
-$01C2  -450   CopySBitMap (11,12)(A0,A1)
-$01C8  -456   OwnBlitter ()
-$01CE  -462   DisownBlitter ()
-$01D4  -468   InitTmpRas (tmpras,buff,size)(A0,A1,D0)
-$01DA  -474   AskFont (rastPort,textAttr)(A1,A0)
-$01E0  -480   AddFont (textFont)(A1)
```

```
-$01E6  -486   RemFont (textFont)(A1)
-$01EC  -492   AllocRaster (width,heigth)(D0,D1)
-$01F2  -498   FreeRaster (planeptr,width,heigth)(A0,D0,D1)
-$01F8  -504   AndRectRegion (rgn,rect)(A0,A1)
-$01FE  -510   OrRectRegion (rgn,rect)(A0,A1)
-$0204  -516   NewRegion ()
-$020A  -522   ** reserviert **
-$0210  -528   ClearRegion (rgn)(A0)
-$0216  -534   DisposeRegion (rgn)(A0)
-$021C  -540   FreeVPortCopLists (viewPort)(A0)
-$0222  -546   FreeCopList (coplist)(A0)
-$0228  -552   ClipBlit (srcrp,srcX,srcY,destrp,destX,destY,sizeX,
               sizeY,minterm)(A0,D0,D1,A1,D2,D3,D4,D5,D6)
-$022E  -558   XorRectRegion (rgn,rect)(A0,A1)
-$0234  -564   FreeCprList (cprlist)(A0)
-$023A  -570   GetColorMap (entries)(D0)
-$0240  -576   FreeColorMap (colormap)(A0)
-$0246  -582   GetRGB4 (colormap,entry)(A0,D0)
-$024C  -588   ScrollVPort (vp)(A0)
-$0252  -594   UCopperListInit (copperlist,num)(A0,D0)
-$0258  -600   FreeGBuffers (animationObj,rastPort,
               doubleBuffer)(A0,A1,D0)
-$025E  -606   BltBitMapRastPort (srcbm,srcx,srcy,destrp,destX,destY,
               sizeX,sizeY,minter)(A0,D0,D1,A1,D2,D3,D4,D5,D6)
```

icon.library

```
-$001E   -30   GetWBObject (name)(A0)
-$0024   -36   PutWBObject (name,object)(A0,A1)
-$002A   -42   GetIcon (name,icon,freelist)(A0,A1,A2)
-$0030   -48   PutIcon (name,icon)(A0,A1)
-$0036   -54   FreeFreeList (freelist)(A0)
-$003C   -60   FreeWBObject (WBObject)(A0)
-$0042   -66   AllocWBObject ()
-$0048   -72   AddFreeList (freelist,mem,size)(A0,A1,A2)
-$004E   -78   GetDiskObject (name)(A0)
-$0054   -84   PutDiskObject (name,diskobj)(A0,A1)
-$005A   -90   FreeDiskObj (diskobj)(A0)
-$0060   -96   FindToolType (toolTypeArray,typeName)(A0,A1)
-$0066  -102   MatchToolValue (typeString,value)(A0,A1)
-$006C  -108   BumbRevision (newname,oldname)(A0,A1)
```

intuition.library

```
-$001E   -30   OpenIntuition ()
-$0024   -36   Intuition (ievent)(A0)
-$002A   -42   AddGadget (AddPtr,Gadget,Position)(A0,A1,D0)
-$0030   -48   ClearDMRequest (Window)(A0)
-$0036   -54   ClearMenuStrip (Window)(A0)
-$003C   -60   ClearPointer (Window)(A0)
-$0042   -66   CloseScreen (Screen)(A0)
-$0048   -72   CloseWindow (Window)(A0)
-$004E   -78   CloseWorkBench ()
-$0054   -84   CurrentTime (Seconds,Micros)(A0,A1)
-$005A   -90   DisplayAlert (AlertNumber,String,Height)(D0,A0.D1)
-$0060   -96   DisplayBeep (Screen)(A0)
-$0066  -102   DoubleClick (sseconds,smicros,cseconds,cmicros)
               (D0,D1,D2,D3)
-$006C  -108   DrawBorder (Rport,Border,LeftOffset,TopOffset)
               (A0,A1,D0,D1)
-$0072  -114   DrawImage (RPort,Image,LeftOffset,TopOffset)(A0,A1,D0,D1)
-$0078  -120   EndRequest (requester,window)(A0,A1)
-$007E  -126   GetDefPrefs (preferences,size)(A0,D0)
-$0084  -132   GetPrefs (preferences,size)(A0,D0)
-$008A  -138   InitRequester (req)(A0)
-$0090  -144   ItemAddress (MenuStrip,MenuNumber)(A0,D0)
-$0096  -150   ModifyIDCMP (Window,Flags)(A0,D0)
-$009C  -156   ModifyProp (Gadget,Ptr,Reg,Flags,HPos,VPos,HBody,VBody)
               (A0,A1,A2,D0,D1,D2,D3,D4)
-$00A2  -162   MoveScreen (Screen,dx,dy)(A0,D0,D1)
-$00A8  -168   MoveWindow (Window,dx,dy)(A0,D0,D1)
-$00AE  -174   OffGadget (Gadget,Ptr,Req)(A0,A1,A2)
-$00B4  -180   OffMenu (Window,MenuNumber)(A0,D0)
-$00BA  -186   OnGadget (Gadget,Ptr,Req)(A0,A1,A2)
-$00C0  -192   OnMenu (Window,MenuNumber)(A0,D0)
-$00C6  -198   OpenScreen (OSArgs)(A0)
-$00CC  -204   OpenWindow (OWArgs)(A0)
-$00D2  -210   OpenWorkBench ()
-$00D8  -216   PrintIText (rp,itext,left,top)(A0,A1,D0,D1)
-$00DE  -222   RefreshGadgets (Gadgets,Ptr,Req)(A0,A1,A2)
-$00E4  -228   RemoveGadgets (RemPtr,Gadget)(A0,A1)
-$00EA  -234   ReportMouse (Window,Boolean)(A0,D0)
-$00F0  -240   Request (Requester,Window)(A0,A1)
-$00F6  -246   ScreenToBack (Screen)(A0)
-$00FC  -252   SCreenToFront (Screen)(A0)
-$0102  -258   SetDMRequest (Window,req)(A0,A1)
-$0108  -264   SetMenuStrip (Window,Menu)(A0,A1)
```

```
-$010E  -270  SetPointer (Window,Pointer,Height,Width,XOffset,YOffset)
              (A0,A1,D0,D1,D2,D3)
-$0114  -276  SetWindowTitles (Window,windowTitle,screenTitle)
              (A0,A1,A2)
-$011A  -282  ShowTitle (Screen,ShowIt)(A0,D0)
-$0120  -288  SizeWindow (Window,dx,dy)(A0,D0,D1)
-$0126  -294  ViewAddress ()
-$012C  -300  ViewPortAddress (Window)(A0)
-$0132  -306  WindowToBack (Window)(A0)
-$0138  -312  WindowToFront (Window)(A0)
-$013E  -318  WindowLimits (Window,minwidth,minheight,maxwidth,
              maxheight)(A0,D0,D1,D2,D3)
-$0144  -324  SetPrefs (preferences,size,flag)(A0,D0,D1)
-$014A  -330  IntuiTextLength (itext)(A0)
-$0150  -336  WBenchToBack ()
-$0156  -342  WBenchToFront ()
-$015C  -348  AutoRequest (WIndow,Body,PText,NText,PFlag,NFlag,W,H)
              (A0,A1,A2,A3,D0,D1,D2,D3)
-$0162  -354  BeginRefresh (Window)(A0)
-$0168  -360  BuildSysRequest (Window,Body,PosText,NegText,Flags,W,H)
              (A0,A1,A2,A3,D0,D1,D2)
-$016E  -366  EndRefresh (Window,Complete)(A0,D0)
-$0174  -372  FreeSysRequest (Window)(A0)
-$017A  -378  MakeScreen (Screen)(A0)
-$0180  -384  RemakeDisplay ()
-$0186  -390  RethinkDisplay ()
-$018C  -396  AllocRemember (RememberKey,Size,Flags)(A0,D0,D1)
-$0192  -402  AlohaWorkbench (wbport)(A0)
-$0198  -408  FreeRemember (RememberKey,ReallyForget)(A0,D0)
-$019E  -414  LockIBase (dontknow)(D0)
-$01A4  -420  UnlockIBase (IBLock)(A0)
```

layers.library

```
-$001E  -30   InitLayers (li)(A0)
-$0024  -36   CreateUpfrontLayer (li,bm,x0,y0,x1,y1,flags,bm2)(A0,A1,
              D0,D1,D2,D3,D4,A2)
-$002A  -42   CreateBehindLayer (li,bm,x0,y0,x1,y1,flags,bm2)(A0,A1,D0,
              D1,D2,D3,D4,A2)
-$0030  -48   UpfrontLayer (li,layer)(A0,A1)
-$0036  -54   BehindLayer (li,layer)(A0,A1)
-$003C  -60   MoveLayer (li,layer,dx,dy)(A0,A1,D0,D1)
-$0042  -66   SizeLayer (li,layer,dx,dy)(A0,A1,D0,D1)
-$0048  -72   ScrollLayer (li,layer,dx,dy)(A0,A1,D0,D1)
-$004E  -78   BeginUpdate (layer)(A0)
```

```
-$0054   -84   EndUpdate (layer)(A0)
-$005A   -90   DeleteLayer (li,layer)(A0,A1)
-$0060   -96   LockLayer (li,layer)(A0,A1)
-$0066  -102   UnlockLayer (li,layer)(A0,A1)
-$006C  -108   LockLayers (li)(A0)
-$0072  -114   UnlockLayers (li)(A0)
-$0078  -120   LockLayerInfo (li)(A0)
-$007E  -126   SwapBitsRastPortClipRect (rp,cr)(A0,A1)
-$0084  -132   WhichLayer (li,x,y)(A0,D0,D1)
-$008A  -138   UnlockLayerInfo (li)(A0)
-$0090  -144   NewLayerInfo ()
-$0096  -150   DisposeLayerInfo (li)(A0)
-$009C  -156   FattenLayerInfo (li)(A0)
-$00A2  -162   ThinLayerInfo (li)(A0)
-$00A8  -168   MoveLayerInFrontOf (layer_to_move,layer_to_be_in_front_
               of)(A0,A1)
```

mathffp.library

```
-$001E   -30   SPFix (float)(D0)
-$0024   -36   SPFlt (integer)(D0)
-$002A   -42   SPCmp (leftFloat,rightFloat)(D1,D0)
-$0030   -48   SPTst (float)(D1)
-$0036   -54   SPAbs (float)(D0)
-$003C   -60   SPNeg (float)(D0)
-$0042   -66   SPAdd (leftFloat,rightFloat)(D1,D0)
-$0048   -72   SPSub (leftFloat,rightFloat)(D1,D0)
-$004E   -78   SPMul (leftFloat,rightFloat)(D1,D0)
-$0054   -84   SPDiv (leftFloat,rightFloat)(D1,D0)
```

mathieeedoubbas.library

```
-$001E   -30   IEEEDPFix (integer,integer)(D0,D1)
-$0024   -36   IEEEDPFlt (integer)(D0)
-$002A   -42   IEEEDPCmp (integer,integer,integer,integer)(D0,D1,D2,D3)
-$0030   -48   IEEEDPTst (integer,integer)(D0,D1)
-$0036   -54   IEEEDPAbs (integer,integer)(D0,D1)
-$003C   -60   IEEEDPNeg (integer,integer)(D0,D1)
-$0042   -66   IEEEDPAdd (integer,integer,integer,integer)(D0,D1,D2,D3)
-$0048   -72   IEEEDPSub (integer,integer,integer,integer)(D0,D1,D2,D3)
-$004E   -78   IEEEDPMul (integer,integer,integer,integer)(D0,D1,D2,D3)
-$0054   -84   IEEEDPDiv (integer,integer,integer,integer)(D0,D1,D2,D3)
```

mathtrans.library

```
-$001E   -30   SPAtan (float)(D0)
-$0024   -36   SPSin (float)(D0)
-$002A   -42   SPCos (float)(D0)
-$0030   -48   SPTan (float)(D0)
-$0036   -54   SPSincos (leftFloat,rightFloat)(D1,D0)
-$003C   -60   SPSinh (float)(D0)
-$0042   -66   SPCosh (float)(D0)
-$0048   -72   SPTanh (float)(D0)
-$004E   -78   SPExp (float)(D0)
-$0054   -84   SPLog (float)(D0)
-$005A   -90   SPPow (leftFloat,rightFloat)(D1,D0)
-$0060   -96   SPSqrt (float)(D0)
-$0066  -102   SPTieee (float)(D0)
-$006C  -108   SPFieee (float)(D0)
-$0072  -114   SPAsin (float)(D0)
-$0078  -120   SPAcos (float)(D0)
-$007E  -126   SPLog10 (float)(D0)
```

potgo.library

```
-$0006    -6   AllocPotBits (bits)(D0)
-$000C   -12   FreePotBits (bits)(D0)
-$0012   -18   WritePotgo (word,mask)(D0,D1)
```

timer.library

```
-$002A   -42   AddTime (dest,src)(A0,A1)
-$0030   -48   SubTime (dest,src)(A0,A1)
-$0036   -54   CmpTime (dest,src)(A0,A1)
```

translator.library

```
-$001E   -30   Translate (inputString,inputLength,outputBuffer,
               bufferSize)(A0,D0,A1,D1)
```

MC68000-Befehlsübersicht

Verwendete Abkürzungen bzw. Symbole:

Label	ein Label bzw. eine Adresse
Reg	Register
An	Adreßregister n
Dn	Datenregister n
Quelle	Quelloperand
Ziel	Zieloperand
<ea>	Adresse oder Register
#n	Direktwert

Mnemonic		Bedeutung
ABCD	*Quelle,Ziel*	Addition zweier BCD-Zahlen
ADD	*Quelle,Ziel*	Binäre Addition
ADDA	*Quelle,An*	Binäre Addition zu einem Adreßregister
ADDI	*#n,<ea>*	Addition mit einer Konstanten
ADDQ	*#n,<ea>*	Schnelle Addition einer Konstanten, welche nur von 0 bis 7 sein darf
ADDX	*Quelle,Ziel*	Addition mit Übertrag im X-Flag
AND	*Quelle,Ziel*	Logisch UND
ANDI	*#n,<ea>*	Logisch UND mit einer Konstanten
ASL	*n,<ea>*	Arithmetische Verschiebung nach links (*2^n)
ASR	*n,<ea>*	Arithmetische Verschiebung nach rechts (/2^n)
Bcc	*Label*	Verzweige bedingt, je nach der Bedingung

BCHG	*#n,<ea>*	Verändere Bit n (aus 0 wird 1 und umgekehrt)
BCLR	*#n,<ea>*	Lösche Bit n
BRA	*Label*	Verzweige unbedingt (ähnlich wie JMP)
BSET	*#n,<ea>*	Setze Bit n
BSR	*Label*	Verzweige in ein Unterprogramm. Ebenso wie bei JSR wird hierbei die Rücksprungadresse auf dem Stack abgelegt und bei RTS zurückgekehrt.
BTST	*#n,<ea>*	Teste Bit n, das Ergebnis steht im Z-Flag
CHK	*<ea>,Dx*	Prüfe ein Datenregister auf Grenzen, löse ggf. die CHK-Instruction-Exception aus
CLR	*<ea>*	Löschen eines Operanden
CMP	*Quelle,Ziel*	Vergleich zweier Operanden
CMPA	*<ea>,An*	Vergleich mit einem Adreßregister
CMPI	*#n,<ea>*	Vergleich mit einer Konstanten
CMPM	*Quelle,Ziel*	Vergleich zweier Speicheroperanden
DBcc	*Reg,Label*	Prüfe Bedingung, dekrementiere und verzweige. Dieser Befehl wird häufig für Schleifen verwendet.
DIVS	*Quelle,Ziel*	vorzeichenrichtige Division des 32-Bit Ziel- durch den 16-Bit Quell-Operanden. Das Ergebnis der Division liegt danach im LO-Wort des Ziels, der Rest der Division im HI-Wort.
DIVU	*Quelle,Ziel*	Division ohne Vorzeichen, ähnlich wie DIVS

EOR	*Quelle,Ziel*	Exklusiv-ODER
EORI	*#n,<ea>*	Exklusiv-ODER mit einer Konstanten
EXG	*Rn,Rn*	Austauschen zweier Registerinhalte (nicht mit SWAP verwechseln!)
EXT	*Dn*	Vorzeichenrichtige Erweiterung auf doppelte Breite
JMP	*Label*	Springe zur Adresse (ähnlich wie BRA)
JSR	*Label*	Springe an ein Unterprogramm. Auch hier wird die Rücksprungadresse auf den Stack gelegt, RTS führt zurück.
LEA	*<ea>,An*	Lade eine effektive Adresse in Adreßregister An
LINK	*An,#n*	Baue Stackbereich auf
LSL	*n,<ea>*	Logische Verschiebung links
LSR	*n,<ea>*	Logische Verschiebung rechts
MOVE	*Quelle,Ziel*	Übertrage einen Wert von Quelle nach Ziel
MOVE	*SR,<ea>*	Übertrage den Statusregister-Inhalt
MOVE	*<ea>,SR*	Übertrage den Statusregister-Inhalt
MOVE	*<ea>,CCR*	Flags laden
MOVE	*USP,<ea>*	Übertrage den User-Stackpointer
MOVE	*<ea>,USP*	Übertrage den User-Stackpointer
MOVEA	<ea>,An	Übertrage einen Wert in das Adreßregister An
MOVEM	*Regs,<ea>*	Übertrage mehrere Register auf einmal
MOVEM	*<ea>,Regs*	Übertrage mehrere Register auf einmal
MOVEP	*Quelle,Ziel*	Übertrage Daten zur Peripherie

MOVEQ	*#n,Dn*	Übertrage schnell eine 8-Bit-Konstante in das Datenregister Dn
MULS	*Quelle,Ziel*	vorzeichenrichtige Multiplikation zweier Worte zu einem Langwort
MULU	*Quelle,Ziel*	Multiplikation ohne Vorzeichen, ähnlich wie MULS
NBCD	*Quelle,Ziel*	Negation einer BCD-Zahl (Neunerkomplement)
NEG	*<ea>*	Negation eines Operanden (Zweierkomplement)
NEGX	*<ea>*	Negation eines Operanden mit Übertrag
NOP		Keine Funktion (No Operation)
NOT	*<ea>*	Inversion eines Operanden
OR	*Quelle,Ziel*	Logisch ODER
ORI	*#n,<ea>*	Logisch ODER mit einer Konstanten
PEA	*<ea>*	Lege eine Adresse auf dem Stack ab
RESET		Peripherie zurücksetzen (Vorsicht!)
ROL	*n,<ea>*	Rotation nach links
ROR	*n,<ea>*	Rotation nach rechts
ROXL	*n,<ea>*	Rotation nach links mit Übertrag ins X-Flag
ROXR	*n,<ea>*	Rotation nach rechts mit Übertrag ins X-Flag
RTE		Rückkehr aus einer Exception
RTR		Rückkehr mit Laden der Flags
RTS		Rückkehr aus einem Unterprogramm (nach BSR oder JSR)

SBCD	*Quelle,Ziel*	Subtraktion zweier BCD-Zahlen
Scc	*<ea>*	Setze ein Byte auf -1, wenn die Bedingung erfüllt ist
STOP		Verarbeitung anhalten (Vorsicht!)evtl. TRAPV-Exception
SUB	*Quelle,Ziel*	Binäre Subtraktion
SUBA	*<ea>,An*	Binäre Subtraktion von einem Adreßregister
SUBI	*#n,<ea>*	Subtraktion einer Konstanten
SUBQ	*#n,<ea>*	Schnelle Subtraktion einer 3-Bit-Konstanten
SUBX	*Quelle,Ziel*	Subtraktion mit Übertrag im X-Flag
SWAP	*Dn*	Vertausche die Registerhälften (die oberen gegen die unteren 16 Bit)
TAS	*<ea>*	Prüfe ein Byte und setze Bit 7
TRAP	*#n*	Springe in eine Exception
TRAPV		Prüfe, ob Überlauf-Flag gesetzt, dann TST <ea> Testen eines Operanden und ggf. setzen von N- und Z-Flag
UNLK	*An*	Baue Stackbereich ab

Stichwortverzeichnis

Absolute Adressierung 33
Abtastrate 157
Addieren von Zahlen 85
Adreßregister 28
Adreßregister direkt 33
Adreßregister indirekt 34
Adressierungsarten 32
Amiga-Betriebssystem 127
Arbeit mit Disketten 74
Arithmetische Operationen 56
ASSEM 67, 68
Assembler 67
Auslesen 164
AUTOKNOB 249

Backup 163
Bedingungscodes 46, 48
Bibliothek 130, 261
Bibliotheken laden 127
Bibliotheksfunktionen 261
Bildschirm öffnen 178
Bildschirmausgaben 140
Binärsystem 17
Binärzahl umrechnen 18
Binärzahlen 93, 97, 99, 102
Bits 16
Boolean-Gadgets 232
Border 227
Byte 16

Chip 115
Chip-RAM 22
CLI-Fenster 140
Clist.library 261
CON: 138
Console.library 262

Container 241, 243
Cursorposition 243
Custom-Chips 20

Datei anlegen 159
Datei löschen 162
Datei öffnen 158
Datei umbenennen 163
Daten lesen 159
Daten schreiben 159
Datenregister 27, 112
Datenregister direkt 33
Datenrichtungs-Register 113
DC 31
Debugger 67
Debugger-Befehle 80
Dezimalsystem 17
Dezimalzahlen 97, 102
Directory 164
Diskettenoperationen 157
Diskettenzugriff 170
Diskfont.library 262
Dos.library 130, 262
Drucker-Ansteuerung 150

Editor 71
Editorfunktionen des K-SEKA 73
EEROM 15
Electric Erasable ROM 15
EPROM 15
Erasable ROM 15
Ereignis-Behandlung 200
EVEN 32
Examine 167
Exception-Programmierung 256
Exception-Routinen 256
Exceptions 41
Exec.library 263

Fast-RAM 22
Fenster öffnen 134, 191
Fenster schließen 138
Feuerknopf 110
FIFO 28
Flags 28
FORMAT 176
Frequenz 117

Gadgets 231
Geschlecht 156
Graphics.library 265

Hardware-Register 15, 123
Hexadezimalsystem 18

Icon.library 267
Images 223
Inhaltsverzeichnis auslesen 164
Interrupts 45
Intuition 177
Intuition-Bibliothek 177
Intuition.library 268

Joystick 108, 126

K-SEKA-Assembler 50, 71
Kopierprogramm 176
KUMA-SEKA 67
Kurvenform 116

Langworte 16
Lautschrift 153
Lautstärke 117, 157
Layers.library 269
Library 130
LIFO 28
Linker 69
Lock-Funktion 166
Logische Operationen 58

Macro definieren 77
Mathffp.library 270
Mathieeedoubbas.library 270
Mathtrans.library 271
Maus 108, 126
MC68000-Befehlsübersicht 272
Mehrfachlinie 228
Menü löschen 203
Menü setzen 203
Menü-Programmierung 203
Menüstruktur 210
Menüzustand 215
Modulation 122
Motor einschalten 173

Narrator 151
Negative Werte 17

OPEN-Funktion 158

Parallelport 111
Piepton 119, 257
Plane-Zeiger 185
Potentiometer 126
Potgo.library 271
Profimat Amiga 67, 70
Programmable Read Only Memory 15
Programmierung des Intuition 179
PROM 15
Proportional-Gadgets 247

Rahmen 227
RAM 14
RAM-Disk 23
Random Access Memory 14
RastPort 221
RAW: 138
Read Only Memory 14
Rechenoperationen 81
Rechteckkurve 116

Registerinhalt 54
Requester 196
ROM 14
Rotieren 59

Schalter 226
Schieberegler 247
Schleife 87
Schließsymbol 200
Screen 178
Screen öffnen 179
Screen schließen 184
Screen verschieben 186
Scrolling 190
Sedezimalsystem 18
Sedezimalzahl umrechnen 18
Sedezimalzahlen 93, 99
Sektoren laden 175
Serielle Ein-/Ausgabe 151
Sirenenton 120
Sondertasten 105
Sortieren einer Tabelle 89
Speicher reservieren 132
Speicher-Adressierung 30
Speicherbereich 132
Spiegelung 105
Sprachausgabe 151
Sprachmodus 156
Sprachsynthesizer 151
Sprechgeschwindigkeit 156
Stack-Bereich 40
Stapelzeiger 28
Statusregister 28
Steuerzeichen 143
Stimmhöhe 156
String-Gadgets 240
Supervisor 40
Supervisor-Modus 253

Tabelle 85
Task 24
Tastatureingaben 146
Textausgabe 221
Timer.library 271
Titelzeile 137
Tonerzeugung 115
Trackdisk.device 170
Translator.library 271

Unmittelbare Adressierung 33
Untermenüs 209
User 40
User-Modus 253
User-Port 200

Vektoren 41
Verschieben 59

OM 15
Worte 16
Write Once Memory 15

Zahlensysteme 17
Zahlensysteme umwandeln 93
Zeichensatz 182
Zeichnung 223

AmigaBASIC für alle. Im ersten Teil werden Sie Schritt für Schritt – und vor allem auf verständliche Art und Weise – in die Programmierung des AMIGA eingeführt: Grafik und Sound gehören genauso dazu wie Datenverwaltung und Statistik. Im zweiten Teil finden Sie alle gelernten Befehle mit Syntax- und Parameterangaben zum schnellen Nachschlagen. Dazu gibt es Programme und Utilities in Hülle und Fülle.

Aus dem Inhalt:

- Das Videotitel-Programm zeigt die OBJECT-Animation
- Das Balken- und Tortengrafik-Programm erklärt die Grafikbefehle
- Das Malprogramm mit Windows, Pulldowns, Mausbefehlen, Füllmustern, Einlesen und Abspeichern von IFF-Bildern
- Das Statistikdaten-Programm hilft, sequentielle Dateien zu verstehen
- Die Datenbank zeigt den Umgang mit relativen Dateien
- Das Sprachutility sorgt für mehr Verständnis bei der Sprachprogrammierung
- Das Synthesizer-Programm führt Sie in die Welt der Töne, Wellenformen und Hüllkurven

Rügheimer, Spanik
AmigaBASIC
Hardcover, 775 Seiten, DM 59,–
ISBN 3-89011-209-9

Amiga Tips und Tricks ist eine riesige Fundgrube für den Amiga-Besitzer. Viele Beispielprogramme in BASIC und C zeigen, wie man die fantastischen Möglichkeiten dieses Superrechners optimal nutzen kann. Und ganz nebenbei lernt man noch eine Menge über den Aufbau des Computers und seine Programmierung.

Aus dem Inhalt:

- Nutzung der wichtigsten Libraries von BASIC aus: Graphic, DOS, Exec, Intuition
- Nutzung der verschiedenen Disk-Fonts in BASIC-Programmen
- Verschiedene Schrifttypen in BASIC-Programmen: Bold, Outline, Shadow
- Zugriff auf das CLI von BASIC aus
- Bewegbare Screens und Windows mit eigenen Titeln
- Intuition in eigenen Programmen nutzen: Autorequest, Guru Meditation
- Gesamte Directory-Struktur ausdrucken
- Ein-/Ausgabehandling: Diskmonitor, Hardcopy von Windows und Screen
- Speicherverwaltung: AllocMem und FreeMem
- Filehandling in C: Anzahl freier Blöcke, File exist Prüfung, Filegröße, Filekommentar, Get Protection Prüfung
- Zugriff auf Intuition am Beispiel eines einfachen Grafikprogramms: Screen, Windows, Menue
- Half Bright und Interlace Modus
- Druckerhandling in C

Weltner/Hornig
AMIGA Tips & Tricks
Hardcover, ca. 320 Seiten, DM 49,–
ISBN 3-89011-211-0

C an einem Wochenende? Durchaus möglich! Mit C für Einsteiger. Ein Einführungskurs, der Ihnen schnell und einfach die wichtigsten Grundlagen dieser Sprache vermittelt. Vom ersten Programm bis hin zu den Routinen in den Bibliotheken. Mit dem gesamten Sprachumfang und den besonderen Features von C. Zahlreiche Tips & Tricks zur Programmierung und eine Beschreibung der beiden Compiler Lattice C und Aztek runden das Ganze ab.

Schaun
Amiga C für Einsteiger
Hardcover, ca. 250 Seiten,
DM 39,–
ISBN 3-89011-107-6

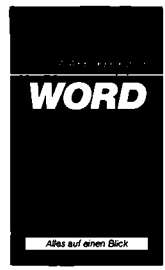
Der Amiga ist das ideale Werkzeug für Kreative – aber einen Film mit ihm erstellen? Warum nicht? Dieses Buch zeigt, wie es geht. Vom Drehbuch bis zum fertigen 3-Minuten-Film. Mit allem, was dazu gehört: Grafik, Sound, Animation und und und. Dabei arbeiten Sie mit den verschiedenen DeLuxe-Programmen, lernen deren fantastische Möglichkeiten kennen und erfahren alles über den professionellen Einsatz von Sounddigitizer und Videorecorder.

Spanik
Amiga – Der Film
Hardcover, ca. 400 Seiten,
ca. DM 59,–
erscheint ca. 8/87
ISBN 3-89011-176-9

In der Regel klappt sie phantastisch, die Arbeit mit dem Computer. Und für Zweifelsfälle hat man ja bereits eine ansehnliche Bibliothek nützlicher Literatur. Doch immer wieder – mitten in der Arbeit – passiert es: Man sucht nach einem bestimmten Kommando. Irgendwo im Handbuch, oder stand es in einem Computermagazin... Der Arbeitsfluß ist unterbrochen. Man versucht sich zu erinnern, durchwühlt den riesigen Literaturberg, sucht einen Hinweis. HILFE. Genau die bekommen Sie von den neuen DATA BECKER Führern. Ein gezielter Griff und Sie haben die gewünschte Information. Hier finden sie umfassend alles auf einem Blick. Zu Ihrem Rechner oder auch zur entsprechenden Software. Das sind die ersten DATA BECKER Führer:

Der DATA BECKER Führer zu MS-DOS

176 Seiten
DM 24,80

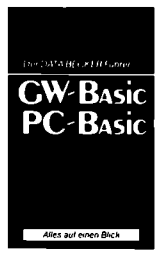


Der DATA BECKER Führer zu WORD

176 Seiten
DM 29,80

Der DATA BECKER Führer zu GW/PC-Basic

160 Seiten
DM 24,80



Der DATA BECKER Führer zu TURBO PASCAL

126 Seiten
DM 24,80

DAS STEHT DRIN:

Wer den AMIGA in Maschinensprache programmieren will, kann nicht nur den komfortablen Befehlssatz des 68000 benutzen, sondern mit den Routinen der AMIGA-Libraries viel Zeit und Arbeit sparen. Wie, das zeigt dieses Buch. Daher finden Sie auch die vielen speziellen Fähigkeiten des AMIGA in diesem Buch wieder – in Maschinensprache programmiert. Und daß Maschinensprache auf dem AMIGA nicht schwer ist, beweist der Autor in diesem Praxis-Buch.

Aus dem Inhalt:

- Der 68000-Prozessor (Befehle, Adressierungsarten, Besonderheiten)
- Aufbau des AMIGA (Speicheraufbau, Betriebssystem, Multitasking)
- AMIGA-DOS (Ein-/Ausgabe, Disketten und Druckeroperationen)
- Intuition (Screens, Windows, Requester, Pull-Down-Menüs, Gadgets)
- Sprachausgabe
- Sound
- Direkte Programmierung der Hardware
- Beschreibung der wichtigsten Assembler (SEKA, PROFIMAT, ASSEM)

UND GESCHRIEBEN HAT DIESES BUCH:

Stefan Dittrich ist Informatikstudent und begeisterter Maschinensprache-Programmierer. Von Anfang an haben ihn die besonderen Fähigkeiten des AMIGA fasziniert. Daß man superschnelle Programme auf dem AMIGA auch sehr einfach und komfortabel schreiben kann, begeistert ihn an diesem Rechner.

ISB N 3-89011-076-2 DM +049.00

DM 49,–
ÖS 382,–
sFr 47,–

DATA BECKER

9 783890 110769

04900

This was brought to you

from the archives of

http://retro-commodore.eu